Wettbewerbsarbeit von Tobia Ochsner

Erstellen von Playlists mithilfe von künstlicher Intelligenz

Kantonsschule Schaffhausen

2018

1 Inhaltsverzeichnis

1 Inhaltsverzeichnis 1
2 Einleitung 3
2.1 Aufbau der Arbeit 3
3 Neuronale Netze 4
3.1 Was sind neuronale Netze? 4
3.2 Neuronale Netze beim Menschen 5
3.3 Aufbau 6
3.4 Training 10
3.5 Entwerfen und Optimieren eines Netzes 18
4 Implementation von neuronalen Netzen 21
4.1 Implementation 21
4.2 Tests anhand MNIST 21
5 Genre-Klassifikation 27
5.1 Vorüberlegungen 27
5.2 Musik 27
5.3 Datengrundlage 30
5.4 Aufbau der getesteten Netze 31
5.5 Resultate 37
5.6 Analyse 38
6 Erstellen von Playlists 43
6.1 Vorüberlegungen 43
6.2 Einordnen von Musik 43
6.3 Datengrundlage 45
6.4 Grundlegender Ansatz 45
6.5 Getestete Fehlerfunktionen 46
6.6 Erste Versuche 48
6.7 Versuche mit Pretraining und Finetuning 49
6.8 Versuche mit Autoencoder 50
6.9 Ensembling 53
6.10 Resultate 54
6.11 Analyse 55
6.12 Erstellen von Playlists 56

6.13 Weiterführende Überlegungen 57
7 Danksagung 59
8 Anhang 60
8.1 Ändern der Gewichte und Schwellenwerte 60
8.2 Probleme mit linearen Aktivierungsfunktionen 64
8.3 Implementation eines neuronalen Netzes 67
8.4 Generieren der Sonogramme 68
8.5 Herleitung der abgeänderten Trio-Fehlerfunktion 68
8.6 Details zu den verwendeten Parametern 71
9 Quellenverzeichnis 74
10 Redlichkeitserklärung 76

2 Einleitung

Musik hat heute für viele von uns einen grossen Stellenwert im Leben. Smartphones und Musikstreamingdienste wie Spotify oder Apple Music ermöglichen uns von überall Zugriff auf eine fast unendlich wirkende Anzahl von Songs und Künstlern. Es war noch nie so einfach, Musik von unbekannteren Künstlern zu geniessen – doch gleichzeitig ist es durch die riesige Auswahl auch nicht unbedingt leichter geworden, neue Musik zu entdecken.

Schon länger hat mich deshalb die Frage beschäftigt, ob es möglich wäre, sich automatisch anhand ein paar gegebener Songs Playlists erstellen zu lassen – und zwar von einem Computer. Optimalerweise würden wir dann, ausgehend von unseren Lieblingssongs, Songs entdecken, die uns ebenfalls gefallen.

Es existieren natürlich verschiedene Wege, wie man dieses Ziel realisieren könnte. Ich habe mich für die Verwendung von sogenannten neuronalen Netzen entschieden. Für viele Beispiele von künstlicher Intelligenz in den letzten Jahren waren neuronale Netze verantwortlich – aus diesem Grund bin ich neugierig, wie diese funktionieren und ob man sie auch im Bereich der Musik einsetzen kann.

2.1 Aufbau der Arbeit

Die Arbeit ist in vier Teile gegliedert:

- Im ersten Teil stelle ich die neuronalen Netze vor und erkläre, wie sie funktionieren.
- Im zweiten Teil implementiere ich ein Programm, mit dem ich eigene neuronale Netze erstellen kann. Mein Programm teste ich anhand der Problemstellung, handgeschriebene Ziffern zu erkennen. Diese Aufgabe bildet für viele den Einstieg in die Welt der neuronalen Netze und eignet sich perfekt, um meine Implementation zu testen.
- Im dritten Teil untersuche ich, wie man neuronale Netze zur Analyse von Songs nutzen kann. Als Vorbereitung auf die Erstellung von Playlists versuche ich zuerst, das Genre eines Songs zu erkennen.
- Im vierten Teil widme ich mich dann schliesslich der Erstellung von Playlists.

3 Neuronale Netze

In diesem Kapitel möchte ich die künstlichen neuronalen Netze vorstellen. Zuerst gebe ich eine kurze Einführung in das Thema. Danach sehen wir uns das biologische Vorbild – unser Gehirn – an. Anschliessend betrachten wir genauer, wie neuronale Netze aufgebaut sind, wie sie funktionieren und wie man sie trainieren kann.

3.1 Was sind neuronale Netze?

Für lange Zeit galten Computer als vergleichsweise dumm: Sie waren zwar bestens geeignet für das Ausführen einer riesigen Anzahl Berechnungen in kurzer Zeit, scheiterten aber an vielen vermeintlich einfachen Aufgaben. An Aufgaben, die wir Menschen beherrschen, ohne uns auch nur anstrengen zu müssen: Am Erkennen von Menschen. Dem Verstehen von Sprache. Dem Lesen von handgeschriebenen Texten. Es musste doch eine Möglichkeit geben, einer Maschine, die Abertausende von Rechenoperationen pro Sekunden ausführen kann, solch einfache Fähigkeiten anzueignen!

Relativ früh kam die Idee auf, zu versuchen, unser Gehirn nachzubauen – es ist schliesslich in der Lage, diese Dinge mit links zu erledigen. Die neuronalen Netze waren geboren. Die ersten Versuche brachten aber nicht den durchschlagenden Erfolg, was vor allem an drei Faktoren lag: der verfügbaren Rechenleistung, dem verfügbaren Speicher und der vorhandenen Menge an Daten.

Weshalb spielt die vorhandene Menge an Daten eine wichtige Rolle? Einer der riesigen Vorteile neuronaler Netze ist, dass sie selbständig lernen, eine Aufgabe zu lösen. Stellen wir uns vor, wir möchten einem Computer beibringen, Bilder der Ziffer Acht zu erkennen. Wir müssen einem neuronalen Netz nun nicht zuerst erklären, dass eine Acht aus zwei aufeinandergestapelten Kreisen besteht. Wir können dem Netz einfach tausende Bilder mit Achten zeigen. Es lernt dann anhand dieser Bilder, wie eine Acht aussieht. Im Zeitalter des Internets ist es heute kein Problem mehr, an diese erforderlichen Daten heranzukommen.

Zusammen mit der gestiegenen Rechenleistung und dem Zerfall der Preise von Speicher führte dies zu einem erneuten Aufschwung der neuronalen Netze in den letzten zehn Jahren. Es entstand ein regelrechter Hype und zahlreiche erstaunliche Ergebnisse wurden damit erzielt. Eine kleine Auswahl:

- 2012 konnten Forscher der Universität Toronto mithilfe eines riesigen neuronalen Netzes bei mehr als einer Million Bilder mit 1000 unterschiedlichen Motiven in 62 % aller Fälle das Motiv erkennen (siehe [1]).
- 2015 entwickelten Forscher ein neuronales Netz, welches Fragen über Objekte auf Bildern beantworten konnte. Es konnte beispielsweise bei Eingabe der Frage «How many chairs are there?» die Anzahl Stühle auf einem Bild zurückgeben. (siehe [2]).

- 2016 gelang es, ein neuronales Netz so zu trainieren, dass es aus einem Schwarzweissbild ein passendes Farbbild erzeugen kann (siehe [3]). Dies sieht dann beispielsweise so aus:

Abbildung 1: Rechts das Farbbild, welches ein neuronales Netz anhand des Schwarzweissbildes links erzeugt hat. (Quelle: https://photogrist.com/wp-content/uploads/2014/05/Ansel-Adams2.jpg und http://demos.algorithmia.com/colorize-photos/)

Doch wie funktionieren neuronale Netze? Vereinfacht gesagt übergibt man einem neuronalen Netz eine Eingabe, es verarbeitet diese und gibt eine Ausgabe zurück:

Abbildung 2: Ein neuronales Netz berechnet aus einer Eingabe eine Ausgabe.

So könnten wir zum Beispiel einem neuronalen Netz das Bild einer handgeschriebenen Ziffer übergeben und es gibt dann zurück, um welche Ziffer es sich handelt:

Abbildung 3: Ein Anwendungsbeispiel ist das Erkennen von handgeschriebenen Ziffern.

Ein neuronales Netz stellt aber keine universelle künstliche Intelligenz dar: Man kann einem Netz jeweils nur eine spezifische Aufgabe antrainieren. Es kann diese Aufgabe zwar oftmals erstaunlich gut erledigen, allerdings wirklich nur diese eine Aufgabe.

3.2 Neuronale Netze beim Menschen

Die Informationen für diesen Abschnitt stammen aus *Markl Biologie* ([4]) und aus *Spektrum der Wissenschaft* ([5], [6] und [7]).

Die künstlichen neuronalen Netze haben ihr Vorbild in den neuronalen Netzen unseres Nervensystems. Diese bestehen aus Nervenzellen, die Neuronen genannt werden und miteinander zu Netzen verbunden sind. Es gibt verschiedene Typen von Neuronen. Wichtig für die künstlichen Netze ist vor allem der allgemeine Aufbau, der bei jedem Neuron im Prinzip gleich ist:

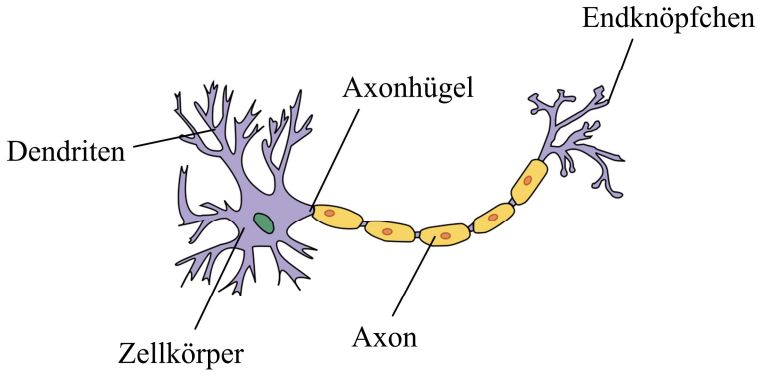


Abbildung 4: Der schematische Aufbau eines Neurons in unserem Körper. (Quelle: https://de.wikipedia.org/wiki/Nervenzelle#/media/File:Neuron_Hand-tuned.svg)

Neuronen besitzen Dendriten, die erregende und hemmende Signale sammeln. Diese Signale sind unterschiedlich stark, ihre Stärke hängt beispielsweise von der Entfernung der Signalquelle ab.

Diese ankommenden erregenden und hemmenden Signale werden zusammengerechnet und an den Axonhügel weitergeleitet. Sobald die Signale dort einen Schwellenwert überschreiten, wird ein elektrisches Signal erzeugt, welches sich entlang des Axons zu den Endknöpfchen hin ausbreitet. An den Endknöpfchen wird das Signal auf andere Zellen, zum Beispiel auf andere Neuronen, übertragen.

Doch wie lernen wir? Der kanadische Psychologe Donald Olding Hebb (1904-1985, siehe [8]) stellte dazu 1949 die Hebbsche Lernregel auf: Nach dieser wird die Übertragung eines Signales von einem Neuron A auf ein Neuron B umso effizienter, je öfters Neuron A Neuron B erregt. Dabei kann sich sowohl die Stärke des Signals als auch die Geschwindigkeit, mit der es weitergeleitet wird, erhöhen. Diese Änderungen können nur wenige Millisekunden, aber auch lebenslänglich beibehalten werden.

Dies sind die wichtigsten Grundlagen über die Funktionsweise der Neuronen in unserem Körper. Im weiteren Text sind der Einfachheit halber mit neuronalen Netzen immer künstliche neuronale Netze und nicht die echten neuronalen Netze gemeint. Dasselbe gilt für die Neuronen.

3.3 Aufbau

In diesem Abschnitt beschreibe ich den grundsätzlichen Aufbau und die Funktionsweise eines einfachen neuronalen Netzes. Die Informationen dazu und zu den Abschnitten 3.4 und 3.5 stammen aus dem Buch *Deep Learning* ([9]) von Ian Goodfellow, Yoshua Bengio und Aaron Courville, dem Online-Buch *Neural Networks and Deep Learning* ([10]) von Michael Nielsen und aus der Wikipedia ([11]).

3.3.1 Aufbau eines neuronalen Netzes

Ein neuronales Netz berechnet aus Eingabewerten Ausgabewerte. Es besteht aus Neuronen, die in Schichten angeordnet sind. Jedes Neuron ist mit den Neuronen der vorhergehenden Schicht verbunden. Die Neuronen der ersten Schicht sind mit den Eingabewerten verbunden:

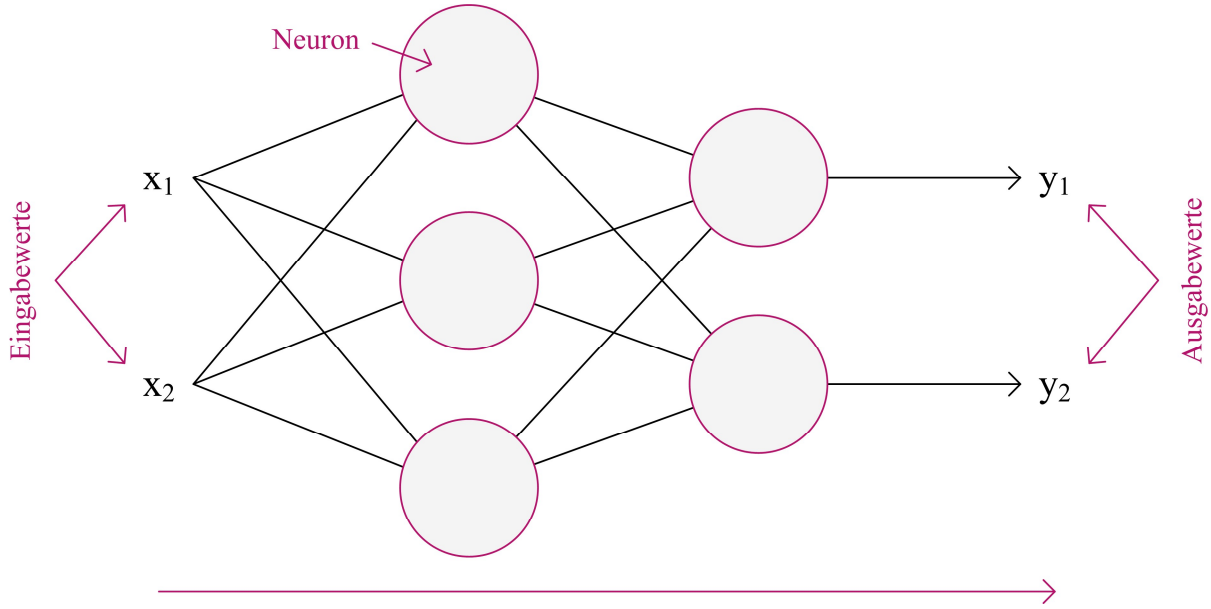


Abbildung 5: Ein neuronales Netz besteht aus Neuronen, welche in Schichten angeordnet sind und mit Verbindungen miteinander verknüpft sind. Jedes Neuron nimmt Werte über die Verbindungen entgegen, verarbeitet sie und gibt einen Wert an die Neuronen der nächsten Schicht weiter.

Die Neuronen nehmen Werte über die Verbindungen entgegen, verarbeiten sie und geben einen Ausgabewert an die Neuronen der nächsten Schicht weiter. Die Neuronen der letzten Schicht geben die Ausgabewerte des Netzes aus.

Die Anzahl Eingabewerte, Ausgabewerte, Schichten und Neuronen können wir je nach Aufgabe beliebig wählen. Beim Beispiel der Ziffernerkennung könnten die Eingabewerte die Helligkeitswerte der einzelnen Pixel des Bildes sein. Die letzte Schicht könnte aus zehn Neuronen für jede Ziffer von Null bis Neun bestehen. Bei einem Bild einer Null sollte der erste Ausgabewert den höchsten Wert annehmen, bei einem Bild einer Eins der zweite Ausgabewert usw:

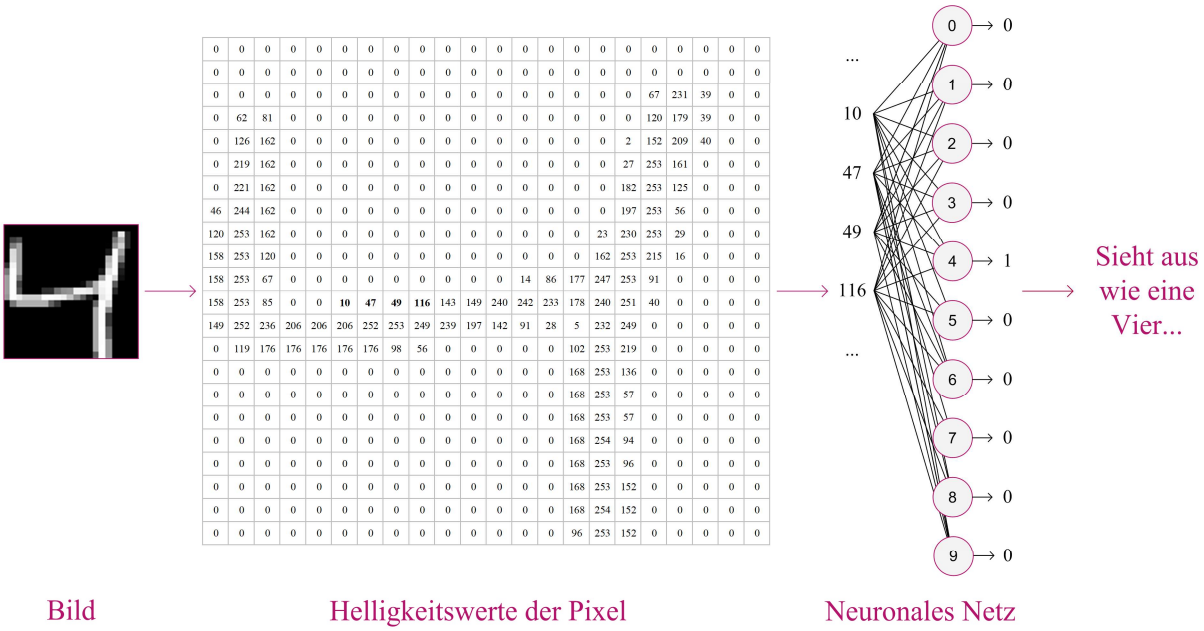


Abbildung 6: Beim Beispiel der Ziffernerkennung könnten die Eingabewerte die Helligkeitswerte der einzelnen Pixel sein. Jedes der zehn Ausgabewerte steht für eine Ziffer.

3.3.2 Neuronen

Ein neuronales Netz setzt sich also wie sein biologisches Vorbild aus Neuronen zusammen. Doch wie funktioniert ein künstliches Neuron?

Ein Neuron nimmt als Eingabewerte die Ausgabewerte der Neuronen der vorhergehenden Schicht über Verbindungen entgegen. Es verarbeitet diese und gibt anschliessend einen Ausgabewert aus:

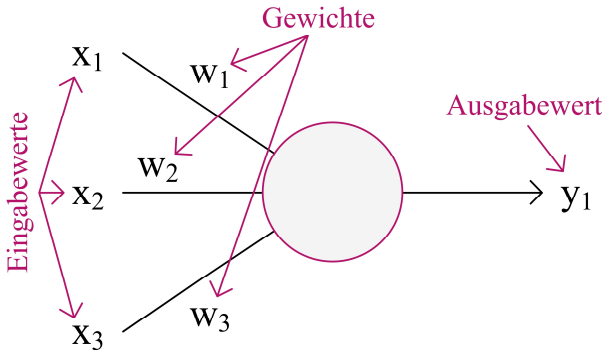


Abbildung 7: Ein Neuron berechnet aus Eingabewerten einen Ausgabewert.

Das Neuron gewichtet jeden Eingabewert mit einer der jeweiligen Verbindung zugeordneten Zahl. Dieses sogenannte Gewicht bestimmt, wie stark der Ausgabewert des Neurons von dem Eingabewert abhängt. Es kann sowohl positiv als auch negativ sein.

Das Neuron summiert alle gewichteten Eingabewerte auf und subtrahiert zusätzlich einen Schwellenwert b, das Ergebnis nennen wir s. Nur falls die aufsummierten, gewichteten Eingabewerte grösser als der Schwellenwert sind, d. h. $\sum_i (x_i\, w_i) > b$, ist s positiv.

$$s = \sum_i (x_i\, w_i) - b. \tag{1}$$

w_i ist dabei der Wert des Gewichtes des Eingabewertes x_i, b ist der Schwellenwert.

Auf s wird nun zusätzlich eine sogenannte Aktivierungsfunktion a(s) angewendet. Deren Ergebnis ist der Ausgabewert des Neurons. Hier lassen sich Parallelen zum echten Neuron ziehen: Die Aktivierungsfunktion übernimmt zusammen mit dem Schwellenwert die Funktion des Axonhügels. Erst wenn die zusammengerechneten Signale beim Axonhügel einen Schwellenwert überschreiten, sendet das Neuron ein Signal aus. Eine entsprechende Aktivierungsfunktion beim künstlichen Neuron könnte also so aussehen:

$$a(s) = \begin{cases} 1 & \text{falls } s > 0, \\ 0 & \text{sonst.} \end{cases} \tag{2}$$

In der Praxis haben sich aber andere Aktivierungsfunktionen als geeigneter herausgestellt. Die populärste Aktivierungsfunktion ist die Rectifier-Funktion (vergl. [12]):

$$a(s) = \begin{cases} s & \text{falls } s \geq 0, \\ 0 & \text{sonst.} \end{cases} \tag{3}$$

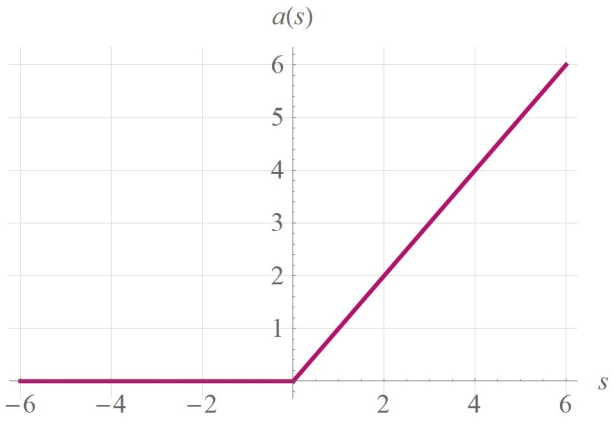


Abbildung 8: Die Rectifier-Funktion (3).

Eine andere verbreitete Aktivierungsfunktion ist die Sigmoid-Funktion (vergl. [13]):

$$a(s) = \frac{1}{1 + e^{-s}}. \tag{4}$$

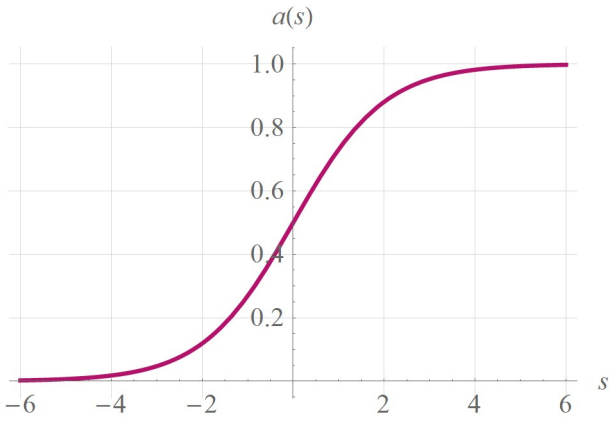


Abbildung 9: Die Sigmoid-Funktion (4).

Die Aktivierungsfunktion ist üblicherweise nichtlinear: Mit einer linearen Aktivierungsfunktion lassen sich bereits einfache Probleme nicht mehr lösen. Genaueres zu den Nachteilen linearer Aktivierungsfunktionen ist im Anhang zu finden.

Zusammengefasst lässt sich der Ausgabewert y eines Neurons wie folgt berechnen:

$$y = a(s) = a(\sum_i (x_i\, w_i) - b). \tag{5}$$

3.3.3 Beispiel

Angenommen, wir haben das unten abgebildete Netz. Es nimmt zwei Eingabewerte entgegen und gibt einen Ausgabewert aus. Die Gewichte lauten wie folgt:

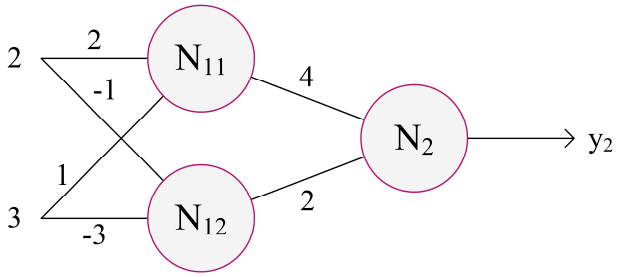


Abbildung 10: Im Beispiel betrachten wir ein neuronales Netz, welches zwei Werte entgegennimmt und einen Wert ausgibt.

Als Aktivierungsfunktion benutzen wir die Rectifier-Funktion (3), alle Schwellenwerte sind 0.

Für die Eingabewerte 2 und 3 können wir die Ausgabewerte der einzelnen Neuronen nun wie folgt berechnen:

$y_{11} = a(2 \cdot 2 + 3 \cdot 1) = a(7) = 7,$
$y_{12} = a\big(2 \cdot (-1) + 3 \cdot (-3)\big) = a(-11) = 0.$

$y_2 = a(y_{11} \cdot 4 + y_{12} \cdot 2) = a(7 \cdot 4 + 0 \cdot 2) = 28.$

Bei der Eingabe von 2 und 3 gibt das Netz also 28 aus.

3.4 Training

Wir haben bisher gesehen, wie ein neuronales Netz aus Eingabewerten Ausgabewerte berechnet. Das Faszinierende an neuronalen Netzen ist, dass sie auch komplexe Eingabewerte wie die Helligkeitswerte eines Bildes analysieren und Ausgabewerte ausgeben können, die uns Informationen über das Bild liefern. Informationen wie zum Beispiel die abgebildete Ziffer. Dies funktioniert allerdings nur, falls den Gewichten und Schwellenwerten passende Werte zugeordnet sind. Doch wie können wir diese passenden Werte finden?

Als erfolgreich bei Problemen wie der Ziffererkennung hat sich die folgende Methode herausgestellt: Man zeigt dem Netz eine grosse Anzahl von Eingabewerten zusammen mit den gewünschten Ausgabewerten. Es versucht dann, aus diesen Beispielen zu lernen und

die Gewichte und Schwellenwerte so anzupassen, dass es bei möglichst vielen der Beispiele passende Ausgabewerte ausgibt. Die genaue Funktionsweise erkläre ich in den folgenden Abschnitten.

3.4.1 Fehlerfunktion

Sobald wir Beispiele gesammelt haben, können wir mit dem Training beginnen. Wir gehen wie folgt vor: Zuerst weisen wir jedem Gewicht und jedem Schwellenwert einen zufälligen Wert zu. Danach wählen wir zufällig eines unserer Trainingsbeispiele aus. Dann ermitteln wir, wie weit die Ausgabe des Netzes bei Eingabe dieses Beispiels von der gewünschten Ausgabe abweicht. Diese Abweichung, auch Fehler genannt, versuchen wir anschliessend zu minimieren.

Was eignet sich als Mass für die Abweichung? Betrachten wir zur Beantwortung dieser Frage ein Netz mit nur zwei Ausgabewerten. Für eine bestimmte Eingabe sollten die Ausgabewerte 1.0 und 0.5 betragen. Die Werte, die das Netz allerdings ausgibt, sind 0.2 und 0.3:

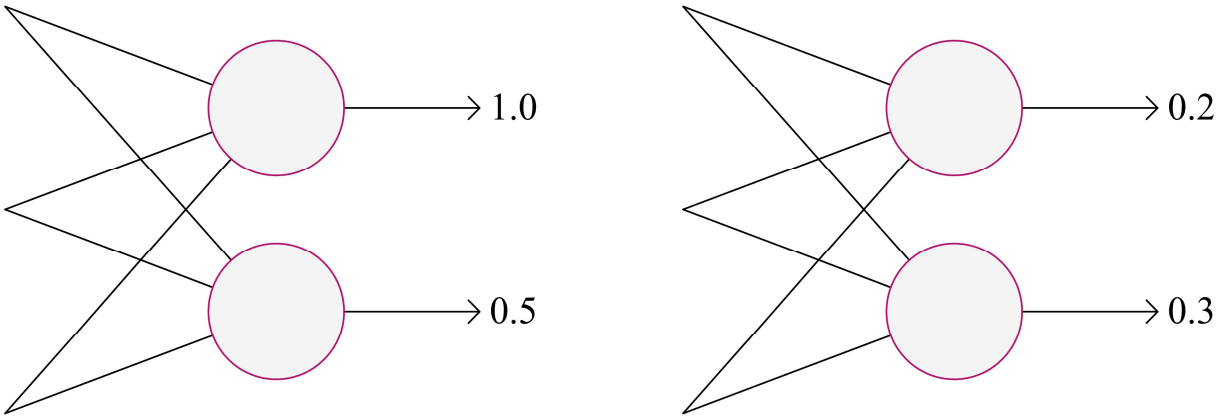


Abbildung 11: Die gewünschten und tatsächlichen Ausgabewerte des Netzes.

Wir können nun das Paar der gewünschten Ausgabewerte wie auch das Paar der tatsächlichen Ausgabewerte als Punkte in einem Koordinatensystem auffassen:

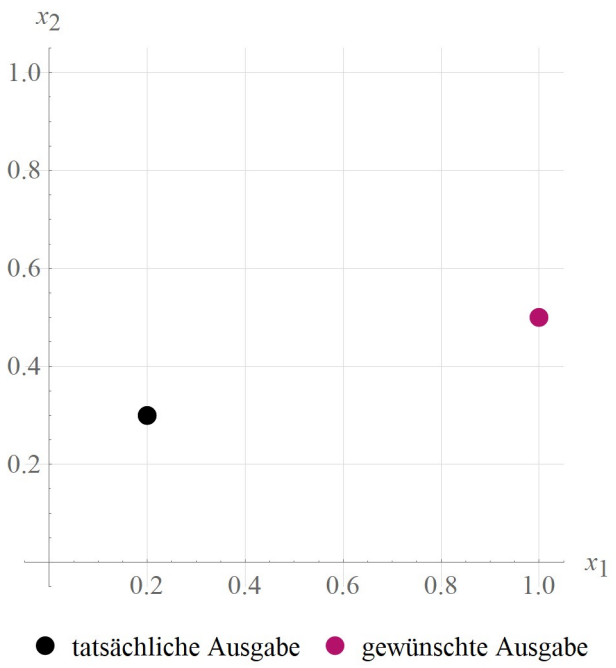


Abbildung 12: Da das Netz nur zwei Werte ausgibt, ist es naheliegend, diese als Koordinaten aufzufassen. Der Fehler können wir dann als Abstand der Punkte der gewünschten und tatsächlichen Ausgabewerte definieren.

Es ist nun naheliegend, den Fehler als Abstand dieser zwei Punkte zu definieren. Dieser lässt sich mit dem Satz des Pythagoras einfach ermitteln:

$$c = \sqrt{(0.2 - 1)^2 + (0.3 - 0.5)^2} \approx 0.8246.$$

Diese Idee können wir auch bei einem Netz mit drei Ausgabewerten verwenden. Die Punkte liegen dann einfach im dreidimensionalen Raum. Den Abstand dieser Punkte können wir ebenfalls mit dem Satz des Pythagoras berechnen:

$$c(y,t) = \sqrt{(y_1 - t_1)^2 + (y_2 - t_2)^2 + (y_3 - t_3)^2}.$$

y sind dabei die Ausgabewerte des Netzes, t die gewünschten Ausgabewerte.

Doch was geschieht bei Netzen mit n Ausgabewerten? Wir können die n Ausgabewerte weiterhin als Koordinaten eines Punktes auffassen. Dieser Punkt liegt dann allerdings in einem n-dimensionalen Raum, welchen wir uns nicht vorstellen können. Wir berechnen den Fehler trotzdem als Abstand der Punkte, die die gewünschten Ausgabewerte und die tatsächlichen Ausgabewerte in diesem n-dimensionalen Raum bilden. Dieser Abstand lässt sich nach dem Satz des Pythagoras wie folgt berechnen:

$$c(y,t) = \sqrt{\sum_i (y_i - t_i)^2}. \tag{6}$$

Dieser Abstand wird auch euklidischer Abstand genannt (vergl. [14]).

In der Praxis wird häufig eine leicht veränderte Version des euklidischen Abstandes eingesetzt. Die Wurzel wird weggelassen und einen Vorfaktor $\frac{1}{2}$ hinzugefügt. Dies hat auf die Eignung als Abstandsfunktion keinen Einfluss, erleichtert aber die Berechnung.

$$c(y,t) = \frac{1}{2}\sum_i (y_i - t_i)^2. \tag{7}$$

Dank dieser Fehlerfunktion können wir also angegeben, wie stark die Ausgabe eines Netzes von der gewünschten Ausgabe abweicht. Diesen Fehler müssen wir nun minimieren.

3.4.2 Anpassen der Gewichte und Schwellenwerte

Unser Ziel ist es, die Gewichte und Schwellenwerte so anzupassen, dass der Fehler bei der Eingabe eines Trainingsbeispiels möglichst klein wird.

Der Ansatz, die Nullstellen der ersten Ableitung zu suchen, ist in diesem Fall allerdings nicht zielführend: Nehmen wir an, wir haben ein neuronales Netz mit nur zwei Gewichten, die wir suchen müssen. Wir können uns die Auswirkung dieser Gewichte auf die Fehlerfunktion genauer ansehen:

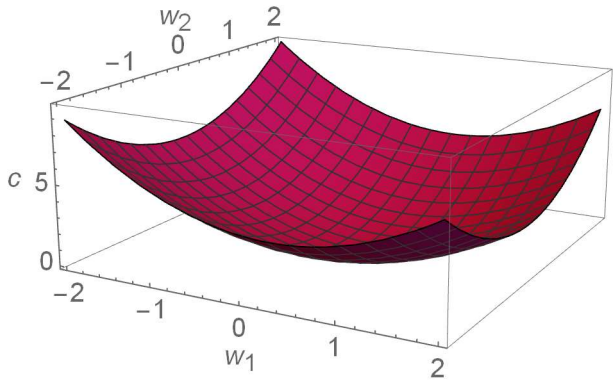


Abbildung 13: So könnte der Graph des Fehlers c in Abhängigkeit zu den Gewichten w_1 und w_2 aussehen. Wir suchen den tiefsten Punkt auf dieser Fläche.

Wir suchen den tiefsten Punkt auf dieser Fläche. An diesem Punkt ist die Ableitung des Fehlers nach den beiden Gewichten null. Wir müssten jetzt also zuerst die Ableitung des Fehlers c nach dem ersten Gewicht w_1 nullsetzen und anschliessend für das zweite Gewicht gleich vorgehen. Das würde zu einem nichtlinearen Gleichungssystem aus zwei Gleichungen und zwei Unbekannten führen. Da man in der Praxis noch viel mehr Gewichte und Schwellenwerte besitzt, würde man ein riesiges, nichtlineares Gleichungssystem aufstellen müssen. Dieses Gleichungssystem hätte eine grosse Anzahl Lösungen und wäre nur numerisch lösbar. Es hat sich herausgestellt, dass es bei einer grossen Anzahl von Variablen bessere Methoden gibt, um ein Minimum zu bestimmen. Eine dieser Methode ist das sogenannte Gradientenverfahren (vergl. [15]):

Wir können uns das Gradientenverfahren so vorstellen, als würden wir bei dichtestem Nebel mitten im Nirgendwo stehen. Wir möchten nun den tiefsten Punkt im Gelände erreichen. Wir ermitteln zuerst, in welche Richtung es am steilsten bergab geht und laufen dann ein paar Meter in diese Richtung. Diese beiden Schritte wiederholen wir so lange, bis wir (hoffentlich) im Tal ankommen.

Konkret wird dieses Verfahren bei neuronalen Netzen wie folgt angewendet:

Angenommen, wir haben ein neuronales Netz und ein Trainingsbeispiel. Das Netz besitzt unter anderem das Gewicht w. Nun übergeben wir dem Netz die Eingabewerte des Beispiels und berechnen die Ausgabewerte. Danach bestimmen wir den Fehler und betrachten, wie dieser von w abhängt. Der Graph zeigt den Fehler c in Abhängigkeit von w:

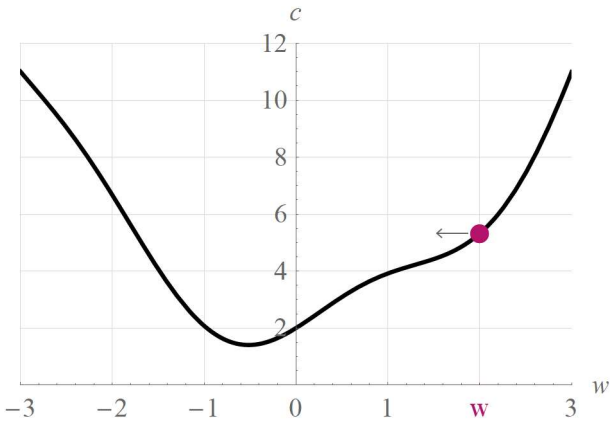


Abbildung 14: So könnte der Graph des Fehlers in Abhängigkeit des Gewichtes w aussehen. Wir suchen das globale Minimum.

Nun passen wir w anhand der Steigung an: Falls die Steigung positiv ist, verkleinern wir w, falls sie negativ ist, vergrössern wir w. Je grösser die Steigung am aktuellen Punkt ist, desto stärker passen wir w an:

$$w := w - \alpha \cdot \frac{\partial c(w)}{\partial w}. \tag{8}$$

α ist dabei die sogenannte Lernrate. Sie bestimmt, wie gross die Schritte sind, die gemacht werden.

Bei den Schwellenwerten können wir analog verfahren.

Zusammengefasst läuft das Training wie folgt ab:

1. Man wählt ein Trainingsbeispiel aus.
2. Die Eingabewerte des Trainingsbeispiels werden dem Netz übergeben und die Ausgabewerte ausgerechnet.
3. Mit (7) wird nun berechnet, wie weit die Ausgabe des Netzes von der gewünschten Ausgabe abweicht.
4. Die Gewichte und Schwellenwerte werden mit (8) so angepasst, dass sich der Fehler verkleinert.
5. Die Schritte 1.-4. werden so lange wiederholt, bis man mit den Fehlern zufrieden ist.

Haben alle Trainingsbeispiele diesen Prozess durchlaufen, nennt man dies eine Epoche. Häufig trainiert man ein Netz für mehrere Epochen. Vor jeder Epoche werden die Trainingsbeispiele zufällig gemischt.

In der Praxis werden die Gewichte und Schwellenwerte nicht nach jedem Beispiel angepasst. Stattdessen werden die Änderungen über eine gewisse Anzahl von Beispielen gemittelt und erst anschliessend angewendet.

3.4.2.1 Mögliche Probleme des Gradientenverfahrens

Obwohl das Gradientenverfahren für das Training von neuronalen Netzen gut funktioniert, besitzt es doch eine Schwachstelle. Wir können mit dieser Methode in den meisten Fällen nicht die allerbestmöglichen Werte für Gewichte und Schwellenwerte finden. Angenommen, der Graph des Fehlers in Abhängigkeit zu einem Gewicht sieht wie folgt aus:

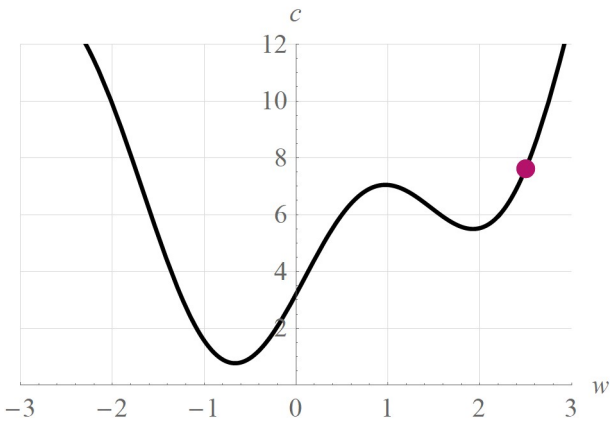


Abbildung 15: w besitzt in diesem Fall den Startwert 2.5. Wir suchen das globale Minimum bei $w \approx -0.7$.

Falls w nun den Startwert $w = 2.5$ besitzt, wird es mit dem Gradientenverfahren in vielen Fällen nicht das globale Minimum bei $w \approx -0.7$ erreichen, sondern nur das lokale Minimum bei $w \approx 1.95$. Mit einer passenden Lernrate können wir zumindest ein wenig gegensteuern.

Mit einer sehr kleinen Lernrate werden wir garantiert ein Minimum erreichen, allerdings kann dies lange dauern und das erreichte Minimum ist höchstwahrscheinlich nicht das globale Minimum:

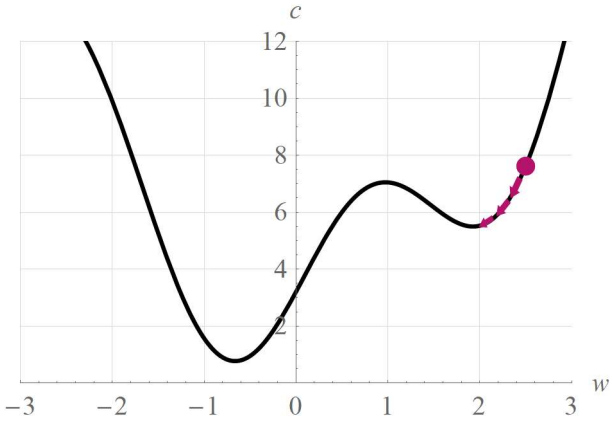


Abbildung 16: Bei einer zu kleinen Lernrate erreichen wir garantiert ein Minimum, jedoch kann dies sehr lange dauern. Das erreichte Minimum ist höchstwahrscheinlich nicht das globale Minimum.

Bei einer zu hohen Lernrate verändert sich der Wert des Gewichtes sehr stark. Es kann vorkommen, dass man ein gutes lokales Minimum wieder verlässt:

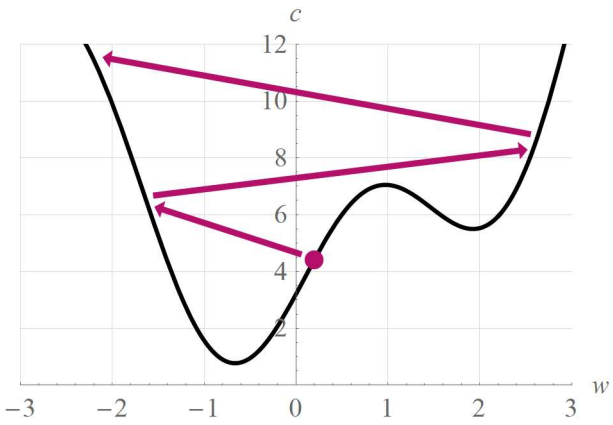


Abbildung 17: Bei einer zu hohen Lernrate verändert sich der Wert des Gewichtes sehr stark. Es kann vorkommen, dass man ein gutes lokales Minimum wieder verlässt.

Weiter unten stelle ich eine Strategie vor, mit der man eine passende Lernrate finden kann.

3.4.2.2 Berechnen der Ableitungen

Beim Gradientenverfahren werden die Ableitungen der Fehlerfunktion nach den Gewichten und Schwellenwerten benötigt. Diese Ableitungen können mithilfe der Kettenregel relativ leicht gefunden werden. Das genaue Vorgehen erkläre ich im Anhang genauer.

3.4.3 Schwierigkeiten beim Training: Unter- und Überanpassung

Eine der grössten Schwierigkeiten beim Training eines Netzes ist die sogenannten Unter- und Überanpassung (siehe auch [16]). Man kann sich diese wie folgt vorstellen:

Vergessen wir die neuronalen Netze kurz und nehmen an, dass wir einen Datensatz mit sechs Datenpunkten haben, welche wie folgt aussehen:

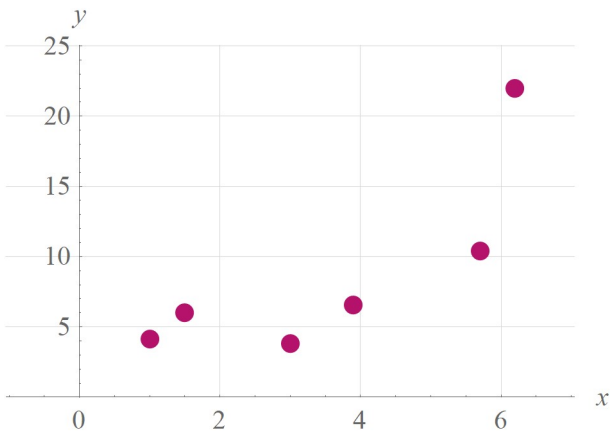


Abbildung 18: Wir versuchen, eine Funktion f(x) zu finden, deren Graphen zu diesen Datenpunkten möglichst gut passt und für Vorhersagen von y-Werten für beliebige x-Werte geeignet ist.

Wir versuchen nun, eine Funktion $f(x)$ zu finden, deren Graphen zu unseren Datenpunkten möglichst gut passt und vor allem für Vorhersagen von y-Werten für beliebige x-Werte geeignet ist. Beginnen wir mit einer einfachen linearen Funktion. Eine passende Gerade können wir beispielsweise mithilfe linearer Regression finden:

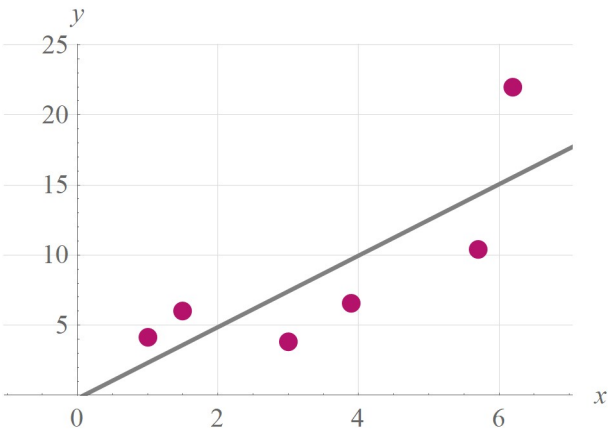


Abbildung 19: Eine lineare Funktion ist zu simpel, als dass sie nützliche Vorhersagen ermöglicht. Dies ist ein Fall einer Unteranpassung.

Wir sehen, dass eine Gerade zu simpel ist, um nützliche Vorhersagen machen zu können. Die ist ein Fall von Unteranpassung – das verwendete Modell ist zu stark vereinfacht, um nützlich zu sein. Bezogen auf neuronale Netze könnte ein Netz zum Beispiel zu wenige Neuronen und Schichten besitzen, um ein Problem zufriedenstellend lösen zu können.

Wir brauchen also eine Funktion, die besser für unsere Zwecke geeignet ist. Wir besitzen sechs Datenpunkte. Wir können somit eine Polynomfunktion fünften Grades finden, die perfekt zu unseren Datenpunkten passt! Doch schon ein kurzer Blick auf den Graphen dieser Funktion zeigt, dass auch diese Funktion für unseren Zweck nicht unbedingt geeignet ist:

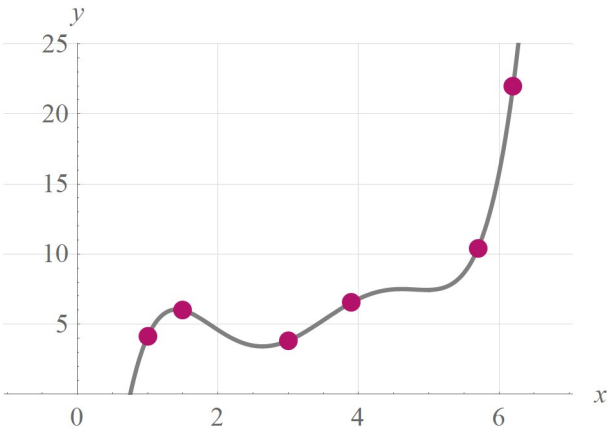


Abbildung 20: Diese Polynomfunktion sagt unsere vorhandenen Datenpunkt perfekt vor. Jedoch liegen andere, unbekannte Datenpunkte wahrscheinlich nicht genau auf diesem Graph. Dies ist ein Fall einer Überanpassung.

Unsere vorhandenen Datenpunkte werden nun perfekt vorhergesagt – doch es ist eher unwahrscheinlich, dass sich andere, unbekannte Datenpunkte wirklich auf diesem Graphen befinden… Wir haben es nun mit einer Überanpassung zu tun: Unser Modell

sagt unsere vorhandenen Daten zwar perfekt vorher, kann aber keine guten Vorhersagen für unbekannte Daten liefern. Bei neuronalen Netzen tritt eine Überanpassung auf, falls zu wenig Trainingsbeispiele zur Verfügung stehen oder das Netz zu gross ist.

In der Praxis ist eine Überanpassung viel häufiger als eine Unteranpassung. Doch wie können wir sie vermeiden oder zumindest abschwächen?

Die einfachste Strategie besteht darin, mehr Trainingsbeispiele zu sammeln oder anhand der vorhandenen Trainingsbeispiele durch geringfügige Anpassungen neue Beispiele zu generieren. Bei Bildern von Ziffern könnten wir die vorhandenen Bilder zufällig drehen, verschieben oder skalieren, um neue Beispiele zu generieren.

Irgendwann reicht das Generieren von zusätzlichen Beispielen nicht mehr aus. Allerdings hat sich gezeigt, dass Netze, deren Gewichte tendenziell klein sind, weniger zur Überanpassung neigen. Tatsächlich lässt sich mit einer Anpassung der Fehlerfunktion die Überanpassung bekämpfen:

Wir fügen der Fehlerfunktion einen weiteren Term hinzu, welcher aus der Summe aller quadrierten Gewichte besteht. Wenn wir während des Trainings nun versuchen, den Fehler zu verkleinern, werden automatisch auch die Gewichte klein gehalten.

$$c(y,t) = \frac{1}{2}\sum_i (y_i - t_i)^2 + \lambda \sum_w w^2. \tag{9}$$

Diese Art der Bekämpfung von Überanpassung wird ℓ^2-Regularisation genannt (vergl. [17]). λ nennt man Regularisationsparameter.

Dieser bestimmt, wie stark die Gewichte verkleinert werden: Ist der Regularisationsparameter zu klein, werden die Gewichte nur leicht verkleinert und eine Überanpassung kann trotzdem auftreten. Ist er zu gross, nähern sich die Gewichte im Laufe des Trainings so stark an 0 heran, dass das Netz die Fähigkeit verliert, überhaupt etwas zu lernen.

3.5 Entwerfen und Optimieren eines Netzes

Die grundlegende Funktionsweise von neuronalen Netzen haben wir in den letzten Abschnitten kennengelernt. Doch wie werden neuronale Netze in der Praxis nun entworfen, was muss man beachten und wie findet man zum Beispiel eine passende Lernrate? Auf diese Themen möchte ich in diesem Abschnitt kurz eingehen.

3.5.1 Trainings- und Testbeispiele

Der Ausgangspunkt ist immer einen Datensatz mit Beispielen, anhand derer das Netz lernen kann. Stellen wir uns vor, wir besitzen ein Datensatz mit Bildern von handgeschriebenen Ziffern. Um verschiedene Netzarchitekturen, Lernraten und Aktivierungsfunktionen testen zu können, müssen wir zuerst einen Weg finden, ein Netz zu bewerten. Dazu können wir bei Klassifikationsproblemen wie der Ziffernerkennung den Anteil der korrekt zugeordneten Beispiele verwenden.

Solange wir ein Netz mit den Bildern testen, mit denen wir es trainiert haben, können wir aber nicht erkennen, ob eine Überanpassung vorliegt. Genau aus diesem Grund teilt man den Datensatz in Trainingsbeispiele und Testbeispiele auf: Wenn wir nun nur mithilfe der Trainingsbeispiele trainieren, können wir anhand der Erkennrate bei den Testbeispielen die Leistung verschiedener Netze zuverlässig vergleichen.

Üblich ist es, 10 % - 30 % des gesamten Datensatzes als Testdatensatz zu verwenden. Es ist wichtig, die Testbeispiele zufällig auszuwählen.

3.5.2 Wahl der Netzarchitektur

Haben wir den Trainings- und den Testdatensatz vorbereitet, kann der eigentliche Aufbau des Netzes beginnen. Wir müssen uns nun entscheiden, wie wir unser Netz aufbauen möchten: Wie viele Schichten soll es besitzen, wie viele Neuronen befinden sich in jeder Schicht?

Bei der Wahl der Netzarchitektur ist es oft sinnvoll, mit einem kleinen Netz anzufangen und es dann, falls man mit der Leistung nicht zufrieden ist, schrittweise zu vergrössern. So kann man zum Beispiel mit nur einer Schicht Neuronen beginnen und dann schrittweise neue Schichten hinzufügen oder die bestehenden vergrössern. Ebenfalls kann ein Aneinanderhängen oder Kombinieren von verschiedenen Netzen in gewissen Fällen sinnvoll sein.

3.5.3 Wahl der Lernrate und des Regularisationsparameters

Hat man sich auf eine Netzarchitektur festgelegt, kann man sich auf die Suche nach einer passenden Lernrate und eines passenden Regularisationsparameters machen.

Bewährt hat sich, mit einer sehr kleinen Lernrate wie $\alpha = 10^{-4}$ zu beginnen und diese dann schrittweise zu verdoppeln oder zu verdreifachen. Leider gibt es keinen Anhaltspunkt, welche Lernrate für welche Probleme geeignet sind – man ist also auf Ausprobieren angewiesen.

Sobald man eine passende Lernrate gefunden hat, kann man einen passenden Regularisationsparameter suchen:

Auch hier hat es sich bewährt, mit einem kleinen λ zu beginnen und dieses dann schrittweise zu erhöhen, bis man zufrieden ist.

3.5.4 Optimierung

Abgesehen von den bisher angesprochenen Möglichkeiten wurden noch viele weitere Wege gefunden, um die Leistung eines neuronalen Netzes zu verbessern. Nicht immer konnte allerdings herausgefunden werden, warum und in welchen Fällen diese Methoden funktionieren. Da sie aber trotzdem häufig den Erfolg massgeblich beeinflussen können, möchte ich zwei Beispiele dieser Optimierungsmöglichkeiten vorstellen.

3.5.4.1 Generieren von weiteren Trainingsbeispielen

Diese Methode habe ich bereits kurz angesprochen. Die Idee ist, durch Veränderungen der vorhandenen Trainingsbeispiele weitere Beispiele zu generieren. Bilder könnte man zum Beispiel leicht skalieren, drehen, verzerren oder spiegeln.

Der Vorteil dieser Methode ist, dass man ohne grossen Aufwand die Anzahl der Trainingsbeispiele vervielfachen kann.

3.5.4.2 Pretraining und Finetuning

Eine gerade im Bereich der Objekterkennung weitverbreitete Methode ist der Einsatz von Pretraining und Finetuning. Bei dieser Methode wird zuerst ein Netz A in einer Aufgabe A trainiert (Pretraining). Danach wird dieses trainierte Netz in ein zweites Netz B integriert, welches anschliessend in einer anderen Aufgabe B trainiert wird (Finetuning).

Ein Beispiel:

Einer der populärsten Datensätze im Bereich der Objekterkennung ist der ImageNet-Datensatz, welcher über zehn Millionen Bilder mit tausend verschiedenen Objekten enthält (siehe [18]). Seit 2010 findet jedes Jahr die *ImageNet Large Scale Visual Recognition Challenge* statt, bei der versucht wird, mithilfe von künstlicher Intelligenz eine möglichst hohe Erkennrate zu erreichen. Als besonders erfolgreich stellten sich neuronale Netze heraus. Einige der erfolgreichsten Netze wurden mitsamt den gefundenen Gewichten und Schwellenwerte veröffentlicht.

Es hat sich nun herausgestellt, dass es für viele andere Aufgaben – unter anderem, aber nicht ausschliesslich im Bereich der Objekterkennung – nützlich ist, eines dieser vortrainierten Netze zu nehmen und nur die letzten Schichten an das eigene Problem anzupassen (verglichen beispielsweise [19]). Diese letzten Schichten werden dann nochmals in der eigentlichen Aufgabe trainiert.

Dieses Vorgehen spart enorm Zeit, da das Netz schon von Anfang an mit einigermassen passenden Gewichten und Schwellenwerten ausgestattet ist und man nicht nochmals mit komplett zufälligen Werten starten muss.

3.5.4.3 Ensembling

Ein weiteres Verfahren, welches zum Beispiel bei der Generierung von Filmvorschlägen (siehe [20]), ist das sogenannte *Ensembling*. Dabei werden mehrere Netze unabhängig voneinander trainiert. Die Ausgabewerte dieser Netze werden dann miteinander kombiniert. Beim Beispiel der Ziffernerkennung könnten wir zuerst mehrere Netze mit denselben Netzgrössen, Lernraten und Regularisationsparametern trainieren. Wenn wir anschliessend bei einem Bild eine Vorhersage treffen wollen, berechnen wir den durchschnittlichen Ausgabewert jedes Ausgabeneurons.

4 Implementation von neuronalen Netzen

Die theoretischen Grundlagen neuronaler Netze sind nun geklärt. Ich zeige in diesem Kapitel, wie ich ein Programm schreibe, mit dem ich neuronale Netze erstellen kann. Um dieses Programm zu testen, trainiere ich damit ein Netz, handgeschriebene Ziffern zu erkennen.

4.1 Implementation

4.1.1 Planung

Zuallererst möchte ich kurz darauf eingehen, mit welcher Programmiersprache ich das Programm implementieren möchte und welche Funktionen es besitzen sollte.

Bei der Programmiersprache habe ich mich relativ schnell auf Python (siehe [21]) festgelegt, da ich bereits Erfahrung mit Python besitze.

Ich möchte die folgenden Funktionen umsetzen:

- Die Anzahl Schichten und Neuronen sollte beliebig gewählt werden können.
- Die Aktivierungsfunktion sollte ebenfalls beliebig gewählt werden können, genau wie die Fehlerfunktion.
- Ein Netz sollte mit dem Gradientenverfahren trainiert werden.
- Die gefundenen Gewichte sollten gespeichert und abgerufen werden können.
- Das Netz sollte so gestaltet werden, dass man mehrere Netze aneinanderhängen kann. Zu diesem Zweck sollte man dem Netz für das Training statt den erwarteten Ausgabewerten auch direkt die Ableitungen der Ausgabewerte nach dem Fehler übergeben können (Mehr Informationen über die Gründe dazu sind im Anhang zu finden.)

4.1.2 Umsetzung

Nachdem die gewünschten Funktionen festgelegt sind, habe ich das Programm in relativ kurzer Zeit entwickeln können. Details zum Programm sind ebenfalls im Anhang beschrieben.

4.2 Tests anhand MNIST

Ob sich mein Programm auch in der Praxis bewähren kann, wird sich bei Experimenten mit dem MNIST-Datensatz zeigen.

4.2.1 Was ist der MNIST-Datensatz?

Der MNIST-Datensatz (siehe [22]) ist eine Sammlung von Bildern von handgeschriebenen Ziffern. Der Datensatz besteht aus 70'000 Bildern mit einer Auflösung von 28 auf 28 Pixeln. Im Folgenden sind einige Beispiele zu sehen:

Abbildung 21: Fünf der 70'000 Bilder handgeschriebener Ziffern des MNIST-Datensatzes.

Da die Bilder klein sind, wird der MNIST-Datensatz oftmals für das Testen von neuartigen Netzarchitekturen, neuen Aktivierungsfunktionen oder anderen Optimierungsmöglichkeiten benutzt. Die bisher besten Netze konnten bei über 99 % aller Bilder die richtige Ziffer erkennen

4.2.2 Experimente

Ich verwende 60'000 der Bilder als Trainingsbeispiele, die verbleibenden 10'000 als Testbeispiele.

Schon bei den ersten Durchläufen zeigt sich, dass mein Programm erfolgreich funktioniert: In relativ kurzer Zeit und ohne grosse Feinabstimmung der Lernrate und Netzarchitektur können die Netze über 90 % der Testbeispiele richtig erkennen. Mit einer optimalen Lernrate und Netzarchitektur erreiche ich eine maximale Erkennrate von rund 98 %. In den folgenden Abschnitten möchte ich einen kurzen Einblick in das Verhalten der Netze geben.

Ich führe alle Experimente viermal durch und mittle anschliessend die Resultate.

4.2.2.1 Anzahl Trainingsbeispiele

Um festzustellen, wie die Erkennrate eines Netzes von der Anzahl der Trainingsbeispiele abhängt, erstelle ich zuerst ein Netz mit $28 \cdot 28 = 784$ Eingabewerten und 10 Ausgabewerten, welches nur aus einer Schicht Neuronen besteht:

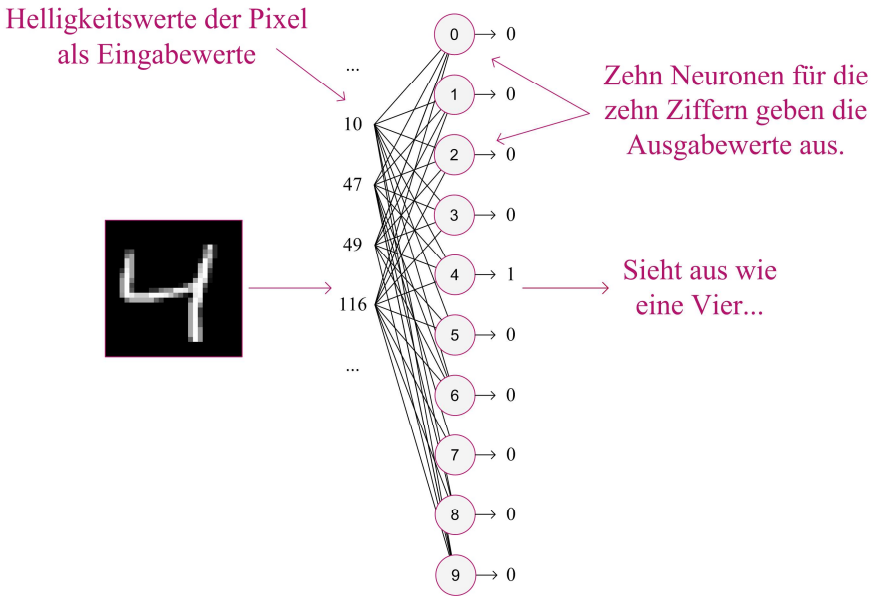


Abbildung 22: Das Netz nimmt die 784 Helligkeitswerte der einzelnen Pixel als Eingabe auf. Es gibt für jede Ziffer einen Wert zwischen 0 und 1 aus.

Als Aktivierungsfunktion setze ich die Sigmoid-Funktion ein, da diese nur Werte zwischen 0 bis 1 ausgeben kann: Das Ausgabeneuron, welches die Ziffer auf dem Bild repräsentiert, sollte 1 ausgeben, alle anderen Ausgabeneuronen 0.

Ich generiere zuerst aus den 60'000 Trainingsbeispielen zufällig eine Auswahl von 100, 1'000 bzw. 10'000 Beispiele und trainiere das Netz anschliessend damit.

Wie erwartet steigt mit der Anzahl verfügbarer Trainingsbeispiele die Erkennrate des Netzes. In der folgenden Tabelle sind die Erkennraten nach 100 Epochen aufgeführt:

Anzahl Trainingsbeispiele	*Erkennrate (Trainingsbeispiele)*	*Erkennrate (Testbeispiele)*
100	37 %	26 %
1000	64 %	60 %
10000	93 %	91 %
60000	91 %	92 %

Interessant ist, dass der Unterschied zwischen der Erkennrate bei den Trainingsbeispielen und den Testbeispielen mit steigender Anzahl Trainingsbeispiele sinkt. Dies zeigt, dass die Tendenz zur Überanpassung bei mehr Trainingsbeispielen abnimmt.

4.2.2.2 Lernrate

Als der folgenden Grafik sehen wir, wie sich verschiedene Lernraten auf das Ergebnis auswirken:

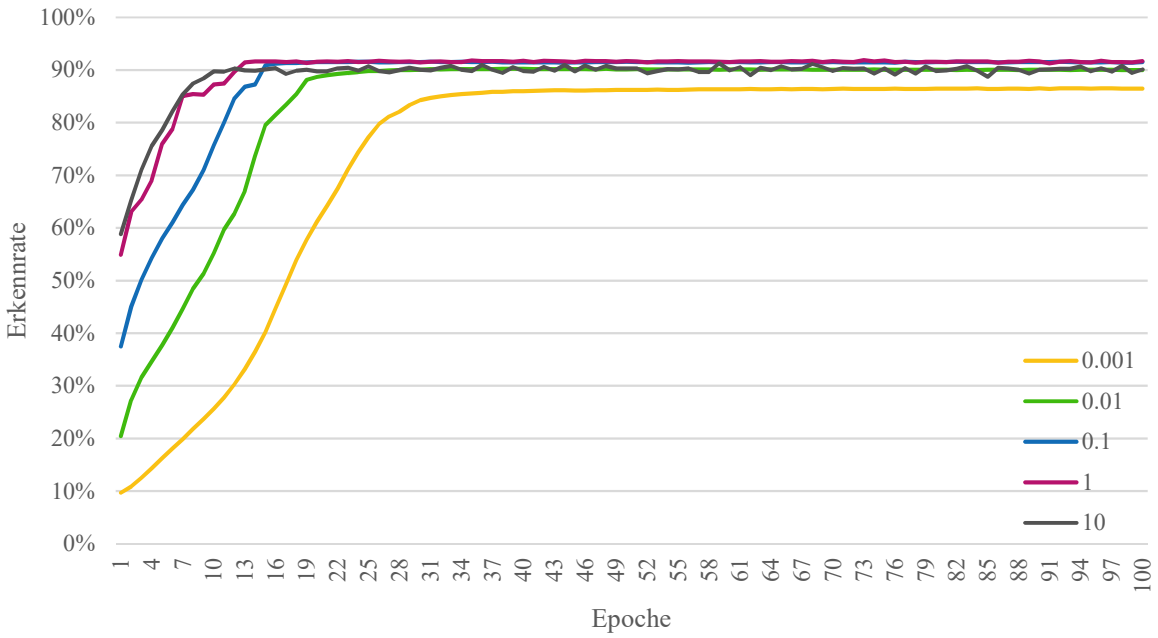


Abbildung 23: Die Erkennrate im Verlaufe des Trainings mit verschiedenen Lernraten. Eine tiefe Lernrate verlangsamt das Training. Eine zu hohe Lernrate kann zu einer tieferen Erkennrate führen.

Lernrate	*Erkennrate (Trainingsbeispiele)*	*Erkennrate (Testbeispiele)*
0.001	86 %	86 %
0.01	89 %	90 %
0.1	92 %	92 %
1	91 %	92 %
10	90 %	90 %

Wir können sehen, dass eine zu kleine Lernrate wie 0.001 das Training verlangsamt, aber auch zu einem schlechteren Resultat führt. Bei einer zu hohen Lernrate wie 10 hingegen steigt die Erkennrate sehr schnell an, kann dann aber sehr stark schwanken oder sogar wieder einbrechen.

4.2.2.3 Grösse des Netzes

Oftmals ist es hilfreich, mehr als nur eine Schicht zu verwenden. Dies zeigt sich auch, wenn ich meinem Netz eine zweite Schicht Neuronen hinzufüge. Ich teste die Leistung des Netzes mit einer zusätzlichen Schicht mit 10, 50, 100 bzw. 200 Neuronen. Ein Netz mit zwei Schichten mit jeweils zehn Neuronen sieht zum Beispiel folgendermassen aus:

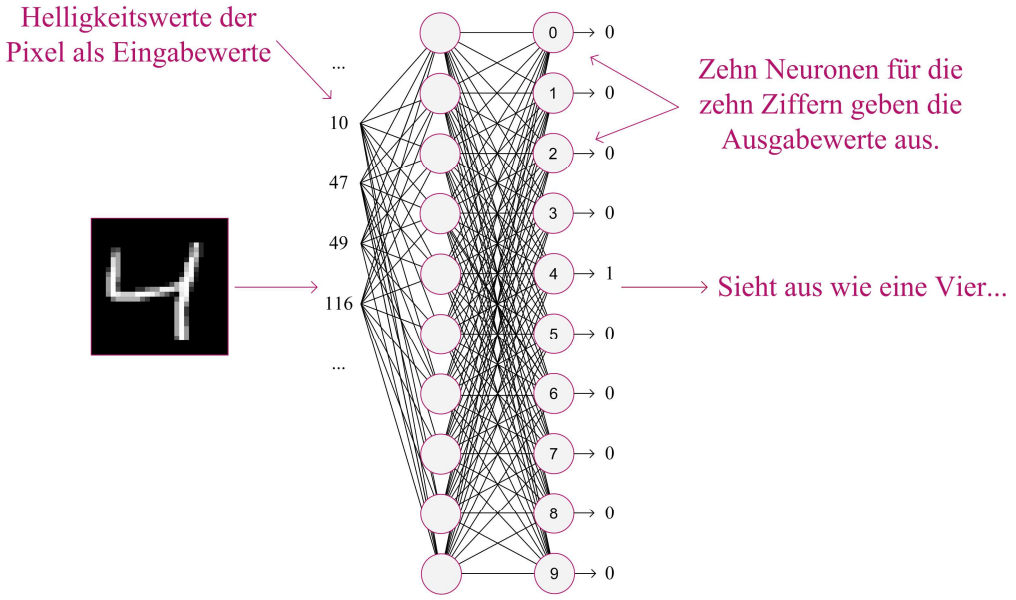


Abbildung 24: Dieses Netz besteht aus zwei Schichten mit je 10 Neuronen.

Dabei bestätigt sich, dass Netze mit mehr Neuronen tendenziell besser lernen können:

Anzahl Neuronen	*Erkennrate (Trainingsbeispiele)*	*Erkennrate (Testbeispiele)*	*Laufzeit (für 100 Epochen)*
1 Schicht à 10 Neuronen	91 %	92 %	4 min 15 s
2 Schichten à 10 Neuronen	94 %	93 %	12 min 05 s

2 Schichten à 50 und 10 Neuronen	99 %	97 %	30 min 49 s
2 Schichten à 100 und 10 Neuronen	99 %	98 %	39 min 34 s
2 Schichten à 200 und 10 Neuronen	99 %	98 %	2 h 10 min 39 s

Allerdings zeigt sich auch, dass irgendwann auch mehr Neuronen nicht unbedingt zu einer besseren Erkennrate führen – auch neuronale Netze stossen an ihre Grenzen.

Die benötigte Laufzeit steigt überproportional an. Man sollte vor dem Hinzufügen neuer Neuronen deshalb immer zwischen Nutzen und Zeitaufwand abwägen: Gerade das Finden einer passenden Lernrate und eines passenden Regularisationsparameters kann einige Testläufe erfordern.

4.2.2.4 Was lernt das Netz?

Es ist faszinierend zu sehen, dass neuronale Netze wirklich funktionieren und sich selbst beibringen können, handgeschriebene Ziffern zu erkennen. Um zu verstehen, was das Netz überhaupt lernt, untersuche ich die Gewichte eines Netzes mit nur einer Schicht Neuronen. Ich erstelle für jedes der zehn Ausgabeneuronen, die ja jeweils eine Ziffer repräsentieren, eine Grafik mit den einkommenden Gewichten. Auf diesen Grafiken lässt sich nun ablesen, auf welche Bildbereiche das Netz bei den unterschiedlichen Ziffern achtet:

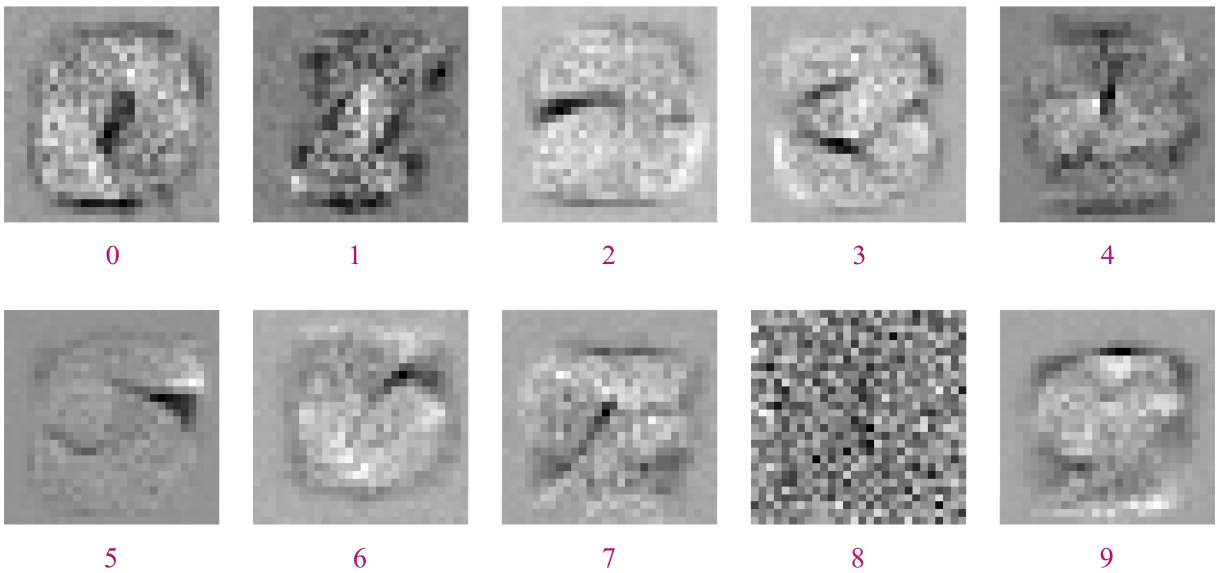


Abbildung 25: Die einkommenden Gewichte der einzelnen Ausgabewerte. Auf den Grafiken lässt sich ablesen, auf welche Bildbereiche das Netz bei den unterschiedlichen Ziffern achtet.

Die hellen Bildbereiche stellen Bereiche dar, die das Netz als wichtig empfindet – Eingabewerte in diesem Bereich werden positiv gewichtet. Eingabewerte in den dunklen Bereichen werden negativ gewichtet. Bei einer Null zum Beispiel werden alle Eingabewerte in einem ringförmigen Bereich positiv gewichtet, die Eingabewerte innerhalb des Kreises in der Mitte hingegen stark negativ.

Interessant finde ich, dass sich bei den meisten Ziffern nachvollziehen lässt, warum das Netz bestimmte Bereiche positiv oder negativ gewichtet. Bei der Acht hingegen scheint kein spezielles Muster zu existieren – sobald sich das Netz sicher ist, dass ein Eingabebild keine der anderen neun Ziffer darstellt, nimmt es an, dass es eine Acht darstellen muss. Dieses Phänomen tritt bei anderen Durchläufen auch bei beliebigen anderen Ziffern auf.

5 Genre-Klassifikation

Obwohl ich am Schluss mithilfe von neuronalen Netzen Playlists erstellen möchte, widme ich mich zuerst einem einfacheren Problem: Ich versuche, einem neuronalen Netz anzutrainieren, das Genre eines Songs zu erkennen. Diese Vorgehensweise hat den Vorteil, dass ich zuerst Erfahrungen mit der Kombination von neuronalen Netzen und Musik sammeln kann, bevor ich mich an die Playlisterstellung wage.

5.1 Vorüberlegungen

Die Erkennung des Genres eines Songs lässt sich grundsätzlich mit der Ziffernerkennung vergleichen: Bei beiden Aufgabestellungen muss ein neuronales Netz eine Eingabe in eine Kategorie einordnen. Somit wird das Netz auch für jedes Genre ein Ausgabeneuron besitzen. Man wird ebenfalls für jedes Genre Trainings- und Testbeispiele benötigen. Es gibt aber immer noch einige Fragen: Wie können wir einen Song einem neuronalen Netz übergeben? Woher bekommen wir die notwendigen Songdateien? Auch beim Aufbau des Netzes werden wir gewisse Dinge beachten müssen.

5.2 Musik

Wie können wir einen Song einem neuronalen Netz übergeben? Um diese Frage zu beantworten, möchte ich zuerst kurz darauf eingehen, was «Musik» überhaupt ist.

Akustische Signale breiten sich in Form von Luftdruckschwankungen (Schallwellen) aus, welche unser Trommelfell in Schwingung versetzen. Diese Schwingungen werden von unserem Ohr weiterverarbeitet.

Das einfachste akustische Signal ist eine Sinusschwingung. Die wahrgenommene Tonhöhe wird von der Frequenz dieser Sinusschwingung bestimmt. Ein Instrument, welches ungefähr eine Sinusschwingung erzeugt, ist die Stimmgabel. Bei vielen Stimmgabeln hat die erzeugte Schwingung eine Frequenz von 440 Hz. Deren Ausschlag-Zeit-Funktion könnte wie folgt aussehen:

Abbildung 26: Der Graph einer Sinusschwingung mit einer Frequenz von 440 Hz. Eine Stimmgabel besitzt beispielsweise eine ähnliche Schwingungskurve.

Wenn verschiedene Instrumente den gleichen Ton spielen, hören wir zwar, dass alle Töne die gleiche Tonhöhe besitzen. Allerdings können wir die Instrumente alleine aufgrund ihres Klanges unterscheiden. Weshalb ist dies so?

Je nach Instrumenten werden neben der Schwingung mit der Grundfrequenz weitere Schwingungen mit anderen Frequenzen erzeugt. Diese zusätzlichen Schwingungen werden zur Schwingung mit der Grundfrequenz addiert. Die Frequenzen und Amplituden der zusätzlichen Schwingungen sind charakteristisch für die einzelnen Instrumente. Wir können sie in einem Amplituden-Frequenz-Diagramm darstellen. Dieses Diagramm nennt man auch Spektrum. Es könnte beispielsweise so aussehen:

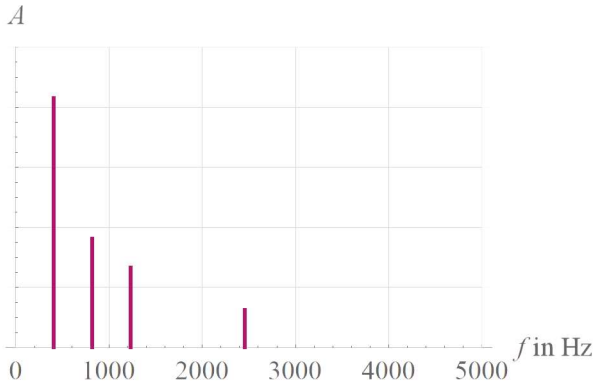


Abbildung 27: Das Spektrum zeigt, welche Sinusfunktionen wir addieren müssen, um zur Ausschlag-Zeit-Funktion der Schwingung zu gelangen.

Die zugehörige Schwingung sieht wie folgt aus:

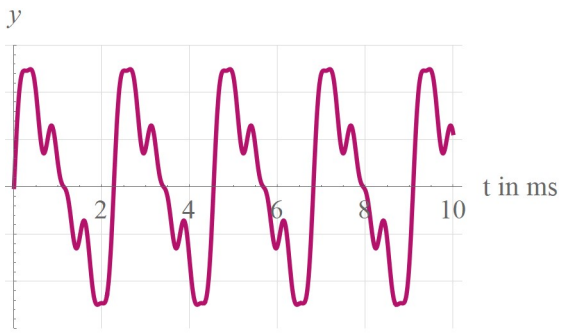


Abbildung 28: Die zum obigen Spektrum gehörende Schwingungskurve.

Das Spektrum zeigt, welche Sinusfunktionen wir addieren müssen, um zur Ausschlag-Zeit-Funktion der Schwingung zu gelangen. Der Weg vom Spektrum zur Ausschlag-Zeit-Funktion ist somit kein Problem.

Der umgekehrte Weg ist ebenfalls möglich. Die dafür verwendete Fourier-Analyse wurde vom französischen Mathematiker und Physiker Jean Baptiste Joseph Fourier (1768-1830, siehe [23]) entwickelt. Sie ist allerdings wesentlich komplizierter als das Addieren von Sinusschwingungen. Ich verzichte deshalb auf eine genauere Erklärung. Ich führe die Fourier-Analyse auch nicht von einem selbstgeschriebenen Programm durch, sondern von der kostenlosen Software sox (siehe [25]).

In der Realität besteht Musik natürlich nicht nur aus genau einem Ton, gespielt von genau einem Instrument. Das Ausschlag-Zeit-Diagramm der Gesamtschwingung wird ungleich komplizierter aussehen und vor allem nicht mehr periodisch sein. Wenn wir uns nur einen Ausschnitt ansehen, können wir ein Signal trotzdem vollständig aus Sinusschwingungen konstruieren.

Das Spektrum für die ersten 10 ms von *Let It Be* von *The Beatles* sieht beispielsweise wie folgt aus:

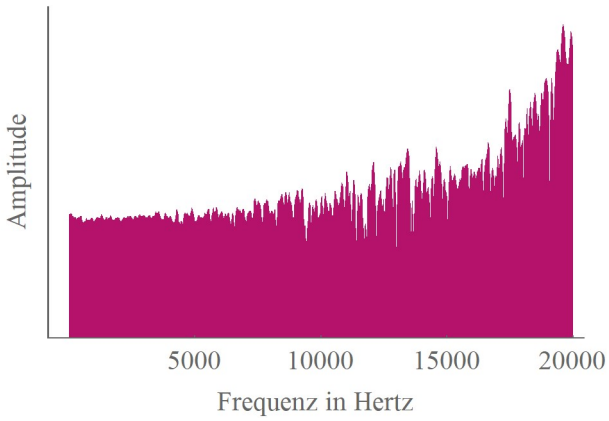


Abbildung 29: Das Spektrum der ersten 10 ms von Let It Be.

Wir können diesen Graphen auch eindimensional darstellen, indem wir die Amplituden als Grauwerte kodieren:

Abbildung 30: Das Spektrum der ersten 10 ms von Let It Be, kodiert als Grauwerte.

Wenn wir dies für alle 10 ms-Abschnitte des Songs durchführen, können wir am Schluss ein sogenanntes Sonogramm erstellen:

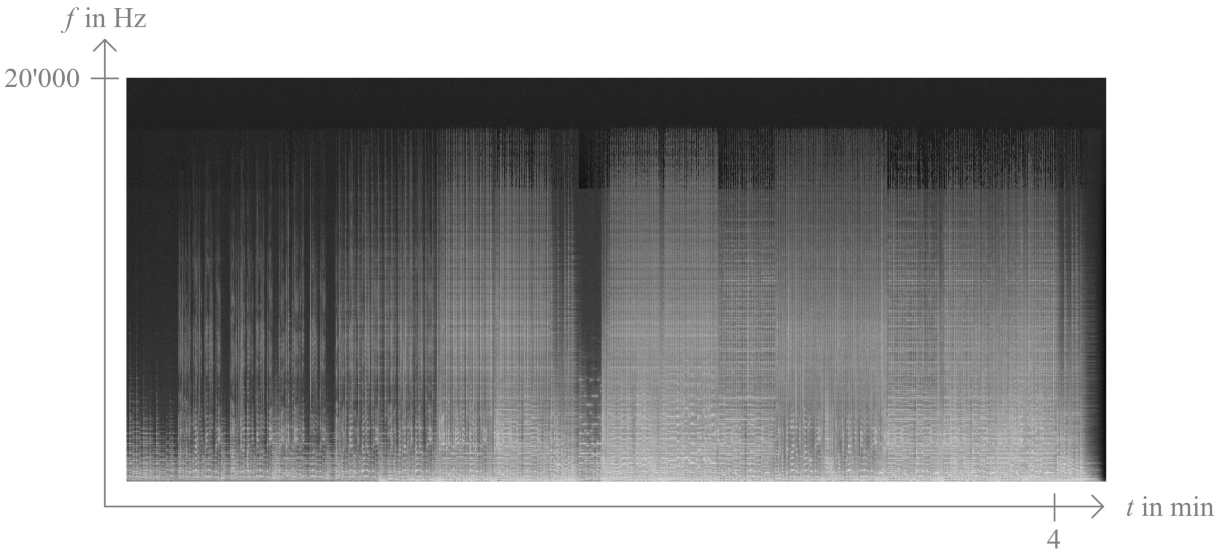


Abbildung 31: Das Sonogramm von Let It Be zeigt das Spektrum im Verlauf der Zeit an.

Auf der horizontalen Achse ist die Zeit, der Verlauf des Songs, aufgetragen. Auf der vertikalen Achse ist die Frequenz aufgetragen. Hohe Frequenzen befinden sich oben, tiefe unten. Die Helligkeitswerte geben die Amplituden der jeweiligen Schwingung an.

Die Helligkeitswerte eines Sonogramms können wir einem neuronalen Netz übergeben. Es wird sich zeigen, ob in dieser Darstellung genug Informationen enthalten sind, um einen Song in sein Genre einordnen zu können.

5.3 Datengrundlage

Am Anfang muss ich zuerst die Daten beschaffen, anhand denen ich das neuronale Netz trainieren kann. Ich benötige zum einen für die wichtigsten Genres eine Liste mit Songs, zum anderen muss ich an die Musikdateien dieser Songs herankommen.

Wie sich herausstellt, kann ich mithilfe des Musikstreamingdienstes Spotify beide Probleme einfach lösen: Abgesehen von den Apps für Computer und Smartphones existiert noch ein weiterer Weg, um auf den riesigen Musikkatalog zuzugreifen: Das Unternehmen bietet mit der *Spotify Web API* eine Schnittstelle an, über die jeder mithilfe eines selbstgeschriebenen Programms Informationen herunterladen kann (siehe [24]).

Da sich auch über die *Spotify Web API* nicht auf das *Genre* eines Songs zugreifen lässt, muss ich einen anderen Weg einschlagen: Spotify bietet tausende von Experten zusammengestellte Playlists an. Ich suche mir nun Playlists heraus, die ausschliesslich Songs eines Genres beinhalten.

Genre	*Playlist*
Klassische Musik	Classical Essentials
Pop	Classic Pop Picks
Rock	Rock This
Rap	Rapcaviar
RnB	Are & Be
Soul	Workday Soul
Jazz	Workday Jazz
Blues	Still Got the Blues
Country	Country Icon
Reggae	Reggae Classics
Folk	Essential Folk
Funk	All Funked Up
Elektronische Musik	Fresh Electronic

Da die Anzahl Songs in jeder Playlist stark variiert, begrenze ich jede Playlist auf die ersten 50 Songs. Somit habe ich $13 \cdot 50 = 650$ Songs zur Verfügung.

Mir ist bewusst, dass diese Lösung ein Kompromiss darstellt, da viele Playlists nicht das gesamte Spektrum eines Genres abdecken. Da der Hauptschwerpunkt meiner Arbeit allerdings nicht auf der Genre-Klassifikation liegt, nehme ich diesen Kompromiss in Kauf.

Auch die Musikdateien kann ich über die Spotify-API herunterladen. Allerdings bietet Spotify nur einen 30-sekündigen Ausschnitt jedes Songs zum Download an.

5.3.1 Erstellung der Sonogramme

Nachdem ich die 650 Songs heruntergeladen habe, erstelle ich aus den mp3-Dateien Sonogramme. Da ich so aber nur 650 Trainingsbeispiele besitze, teile ich jedes Sonogramm in 10 Abschnitte zu je 3 Sekunden auf. Dadurch kann ich plötzlich 6'500 Trainingsbeispiele für das Training verwenden.

5.4 Aufbau der getesteten Netze

Wie soll ich ein Netz zur Genre-Klassifikation am sinnvollsten aufbauen? Die einfachste Möglichkeit wäre, das Sonogramm direkt in Form der Helligkeitswerte einem Netz zu übergeben. Dies ist allerdings nicht die beste Variante: Ich besitze nur jeweils einen 3-sekündigen Ausschnitt eines Songs. Für die Erkennung des Genres sollte es keine Rolle spielen, welche drei Sekunden ausgewählt werden. Wenn wir allerdings vom selben Song zwei leicht verschobene Ausschnitte nehmen und für beide das Sonogramm generieren, sehen die Eingabewerte völlig unterschiedlich aus:

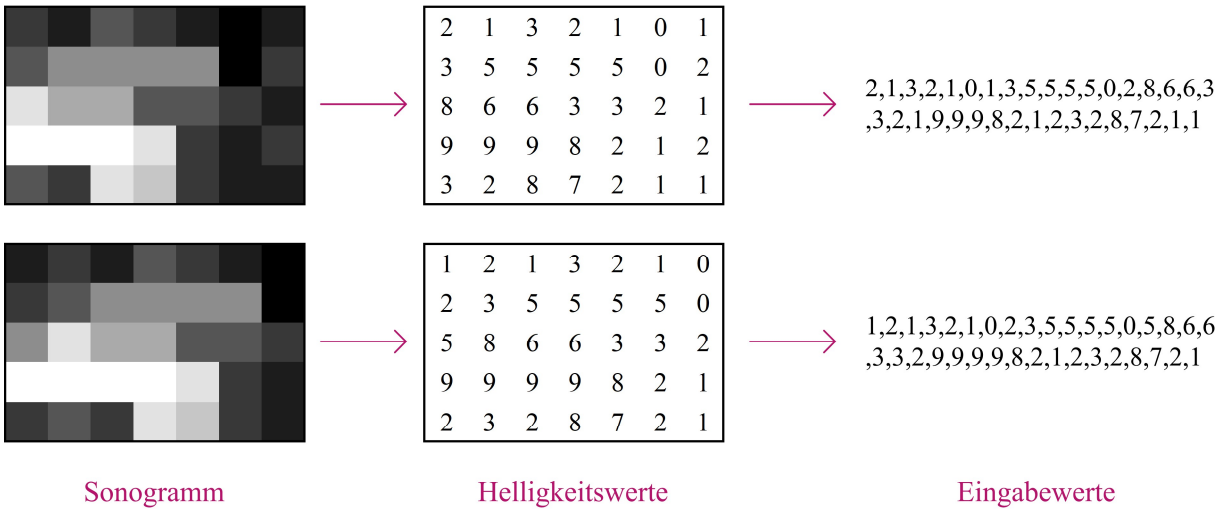


Abbildung 32: Wir übergeben einem Netz direkt das Sonogramm. Wenn wir allerdings vom selben Song zwei leicht verschobene Ausschnitte nehmen, sehen die Eingabewerte völlig unterschiedlich aus.

Deshalb lasse ich mich von [26] und [27] inspirieren, nicht die gesamten drei Sekunden direkt in ein Netz einzuspeisen und stattdessen immer nur einen kleinen Ausschnitt des Sonogramms zu analysieren. Ich habe elf unterschiedliche Netzarchitekturen getestet. Im Folgenden möchte ich drei dieser Netzarchitekturen vorstellen:

5.4.1 Netz 1

Zuerst möchte ich die allgemeine Idee erklären. Wir sehen uns der Einfachheit halber im Folgenden ein stark vereinfachtes Sonogramm an:

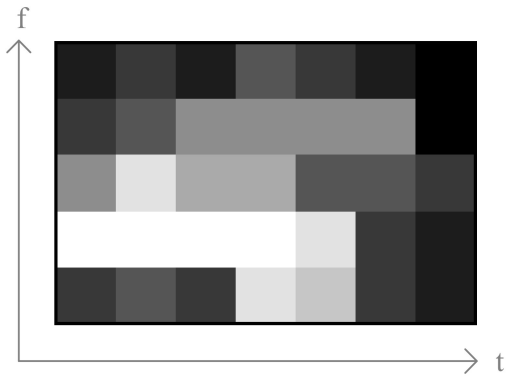


Abbildung 33: Das Sonogramm, welches wir uns als Beispiel ansehen.

Betrachten wir zuerst nur die erste Spalte. Wir übergeben die Helligkeitswerte der Pixel an ein neuronales Netz, welches fünf Werte als Eingabewerte annimmt und genau einen Wert ausgibt:

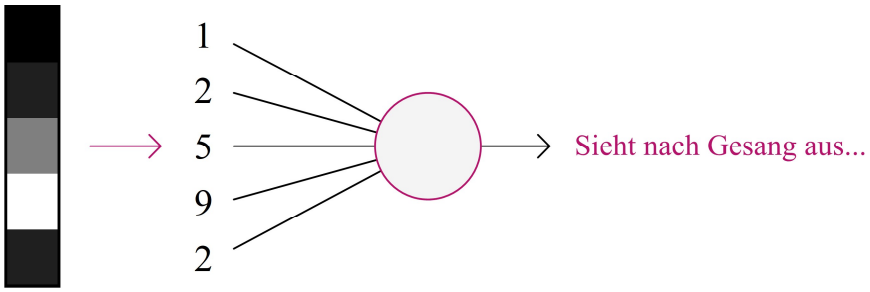


Abbildung 34: Wir übergeben die Helligkeitswerte der Pixel einer Spalte an ein neuronales Netz, welches fünf Werte als Eingabewerte annimmt und genau einen Wert ausgibt.

Wir können uns das Netz als Indikator für eine bestimmte Eigenschaft vorstellen. Beispielsweise könnte ein hoher Ausgabewert andeuten, dass in diesem Songabschnitt Gesang vorkommt.

Demselben Netz übergeben wir die Helligkeitswerte der anderen sieben Spalten. Somit können wir nun sehen, wie sich der Ausgabewert mit der Zeit ändert. Dies könnte vielleicht wie folgt aussehen:

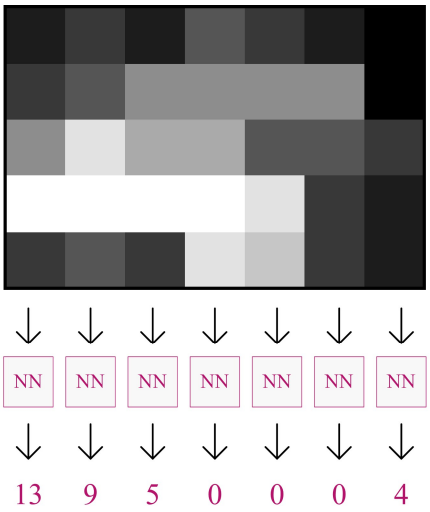


Abbildung 35: Übergeben wir jede Spalte unseres Sonogramms dem Indikatornetz, sehen wir, wie sich der Ausgabewert mit der Zeit ändert.

Nun können aber selbst wir Menschen anhand nur eines Kriteriums nicht bestimmen, welchem Genre ein Song angehört: Gesang beispielsweise ist ein Merkmal der meisten Musikgattungen. Aus diesem Grund verwenden wir nicht nur ein neuronales Netz als Indikator, sondern 50 davon. Diese 50 Netze sind alle gleich aufgebaut. Die Gewichte und Schwellenwerte dieser Netze sind aber unabhängig voneinander – jedes Netz kann also einen anderen Indikator darstellen. Die Ausgabewerte dieser Indikatoren können wir in einer Indikatorkarte darstellen, welche beispielsweise so aussehen könnte:

								Ø
Indikator 1	13	9	5	0	0	0	4	4.4
Indikator 2	4	6	10	10	10	10	0	7.1
Indikator 3	0	0	0	0	0	0	0	0.0
Indikator 4	1	2	1	3	2	1	0	1.4
Indikator 5	5	3	1	9	9	0	0	3.9
Indikator 6	5	8	6	6	3	3	2	4.7
...								
Indikator 50	4	6	4	16	14	4	2	7.1

Abbildung 36:Die Ausgabewerte der 50 Indikatornetze können wir in einer Indikatorkarte darstellen.

Uns interessiert aber nur, ob eine bestimmte Eigenschaft in einem Songausschnitt vorkommt – ob dies nun am Schluss, in der Mitte oder am Ende der Fall ist, spielt für das Erkennen des Genres keine Rolle. Aus diesem Grund berechnen wir für jeden Indikator den durchschnittlichen Ausgabewert über den ganzen Songausschnitt. Diese 50 Durchschnittswerte sollten nun relativ unabhängig vom gewählten Ausschnitt sein. Somit können wir sie einem weiteren neuronalen Netz übergeben, welches für jedes Genre ein Ausgabeneuron besitzt.

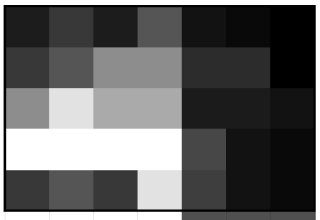
Ich übergebe die gemittelten Ausgabewerte der Indikatornetze an ein Netz bestehend aus zwei Schichten mit 100 und 13 Neuronen:

Abbildung 37: Der Aufbau meines ersten Netzes.

5.4.2 Netz 2

Unsere Sonogramm-Ausschnitte besitzen eine Breite von 40 Pixeln. Eine Spalte repräsentiert somit einen Zeitabschnitt von $\frac{3\text{ s}}{40} = 0.075\text{ s} = 75\text{ ms}$. Wir Menschen benötigen nicht länger als 300 ms, um das Genre eines Songs relativ zuverlässig bestimmen zu können (siehe [32]) – somit könnten bereits in vier Spalten der Sonogramme wertvolle Informationen vorhanden sein. Diese Tatsache nutzen wir im Folgenden aus. Ich möchte die Idee auch zuerst an unserem kleinen Sonogramm vorstellen.

Beim Netz 2 handelt es sich grundsätzlich um das Netz 1 mit einer zusätzlichen Schicht. Auch Netz 2 besitzt zuerst Indikatornetze, nur berechnen wir von der Indikatorkarte aus noch nicht den Durchschnitt. Stattdessen erstellen wir ein neues Netz mit $4 \cdot 50 = 200$ Eingabewerten und einem Ausgabewert. Diesem Netz übergeben wir die ersten vier Spalten der Indikatorkarte:



Indikator 1	13	9	5	0	0	0	4
Indikator 2	4	6	10	10	10	10	0
Indikator 3	0	0	0	0	0	0	0
Indikator 4	1	2	1	3	2	1	0
Indikator 5	5	3	1	9	9	0	0
Indikator 6	5	8	6	6	3	3	2
...							
Indikator 50	4	6	4	16	14	4	2

Sieht nach Kopfstimme aus...

Abbildung 38: Wir übergeben jeweils vier Spalten des Indikatornetzes des ersten Netzes einem weiteren Netz.

Wir können dieses Netz nun wiederum als Indikator auffassen, der vielleicht bestimmte Gesangsarten erkennt. Wir wenden dieses Netz jetzt wieder auf alle Spalten an, indem wir den Ausschnitt immer um eine Spalte verschieben. Hier verwenden wir ebenfalls mehr als nur ein Indikatornetz und auch hier können wir aus den Ausgabewerten eine Indikatorkarte erstellen:

Indikator 1	13	9	5	0	0	0	4
Indikator 2	4	6	10	10	10	10	0
Indikator 3	0	0	0	0	0	0	0
Indikator 4	1	2	1	3	2	1	0
Indikator 5	5	3	1	9	9	0	0
Indikator 6	5	8	6	6	3	3	2
...							
Indikator 50	4	6	4	16	14	4	2

					Ø
Indikator 1	2	1	1	0	1.0
Indikator 2	0	0	1	0	0.3
Indikator 3	2	1	0	0	0.8
Indikator 4	0	1	2	3	1.5
Indikator 5	5	9	5	9	7.0
Indikator 6	9	8	2	3	5.5
...					
Indikator 50	0	1	0	2	0.8

Abbildung 39: Die Indikatorkarte des zweiten Netzes.

Wir mitteln die Ausgabewerte der Indikatornetze der zweiten Schicht und übergeben diese Durchschnittswerte einem weiteren Netz, welches schliesslich das vermutete Genre ausgibt.

Netz 2 besteht aus folgenden Teilnetzen:

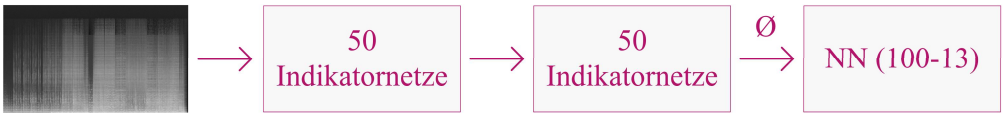


Abbildung 40: Das zweite Netz.

5.4.3 Netz 3

Netz 3 ist am Anfang und am Schluss identisch aufgebaut wie Netz 2. Es besitzt aber noch eine weitere Schicht mit Indikatornetzen, die jeweils vier Spalten der zweiten Indikatorkarte als Eingabewerte annehmen:

Indikator 1	13	9	5	0	0	0	4
Indikator 2	4	6	10	10	10	10	0
Indikator 3	0	0	0	0	0	0	0
Indikator 4	1	2	1	3	2	1	0
Indikator 5	5	3	1	9	9	0	0
Indikator 6	5	8	6	6	3	3	2
...							
Indikator 50	4	6	4	16	14	4	2

Indikator 1	2	1	1	0
Indikator 2	0	0	1	0
Indikator 3	2	1	0	0
Indikator 4	0	1	2	3
Indikator 5	5	9	5	9
Indikator 6	9	8	2	3
...				
Indikator 50	0	1	0	2

		Ø
Indikator 1	1	1
Indikator 2	1	1
Indikator 3	9	9
Indikator 4	0	0
Indikator 5	0	0
Indikator 6	2	2
...		
Indikator 50	0	0

Abbildung 41: Die Indikatorkarte des dritten Netzes.

Dem neuronalen Netz am Schluss werden die gemittelten Ausgabewerte der Indikatornetze der dritten Schicht übergeben. Netz 3 besteht also aus folgenden Teilnetzen:

Abbildung 42: Das dritte Netz.

5.5 Resultate

Nachdem ich die verschiedenen Netze implementiert habe, lasse ich die ersten Tests laufen. Obwohl das Training im Prinzip erfolgreich verläuft, zeigt es doch die Grenzen meines selbstgeschriebenen Programmes auf: Das Training dauert teilweise länger als einen Tag. Gerade für das Ausprobieren verschiedener Netzarchitekturen oder Lernraten ist dies bedeutend zu lange.

Mir gelingt es zwar, mithilfe einiger Tricks die Laufzeit etwas zu verkürzen. Schlussendlich entscheide ich mich trotzdem, nach Alternativen zu suchen. Diese finde ich in Keras ([28]): Mithilfe von Keras kann man neuronale Netze in Python entwerfen und trainieren. Der Vorteil gegenüber meinem Programm ist seine schnellere Laufzeit.

5.5.1 Vergleich der verschiedenen Netzarchitekturen

Sehen wir uns die Erkennrate der drei betrachteten Netze nach dem Training an. Ich trainiere alle Netze für 300 Epochen. 30 % der Songs verwende ich als Testbeispiele. In jeder Epoche übergebe ich dem Netz von jedem Song zwanzig zufällig gewählte, drei Sekunden lange Ausschnitte. Ich führe jedes Training vier Mal durch.

Die maximale Erkennrate der Testbeispiele liegt bei 74 %:

Netz	*Erkennrate (Trainingsbeispiele, maximal)*	*Erkennrate (Testbeispiele, maximal)*	*Laufzeit (für 300 Epochen)*
Netz 1	85 %	65 %	50 min 18 s
Netz 2	89 %	70 %	1 h 37 min 15 s
Netz 3	92 %	74 %	1 h 45 min 22 s

Dabei zeigt sich, dass jede zusätzliche Schicht an Indikatornetzen eine Verbesserung hervorbringt, sich allerdings auch die Laufzeit verlängert. Es liegt nun nahe, weitere Indikatorschichten einzufügen. Dies steigert die Erkennrate allerdings nicht, im Gegenteil: Mit einer vierten Schicht Indikatornetze liegt die maximale Erkennrate bei 61 %, mit einer fünften bei 53 %.

Interessant ist auch ein zweiter Aspekt: Wir können eine starke Überanpassung beobachten, die Erkennrate der Testbeispiele liegt weit unter der der Trainingsbeispiele. Ich vermute, dass dies an der geringen Anzahl Songs liegt und man hier mit mehr Trainingsbeispielen eine Verbesserung erzielen könnte.

5.6 Analyse

In diesem Abschnitt analysiere ich das trainierte Netz 3. Zuerst sehe ich mir an, wie gut die Erkennrate der einzelnen Genres ist. Danach untersuche ich, was die einzelnen Indikatornetze gelernt haben.

Zuerst untersuche ich, als was Songs der verschiedenen Genres klassifiziert werden. Ich betrachte dabei sowohl die Trainings- als auch die Testbeispiele:

	Vermutetes Genre des Songs												
Genre des Songs	*Pop*	*Rock*	*Reggae*	*Rap*	*Country*	*Blues*	*Jazz*	*Soul*	*Klassik*	*Elektronisch*	*RnB*	*Folk*	*Funk*
Pop	**78%**	0%	0%	11%	0%	0%	0%	0%	0%	0%	11%	0%	0%
Rock	10%	**70%**	0%	0%	0%	20%	0%	0%	0%	0%	0%	0%	0%
Reggae	0%	0%	**100%**	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Rap	0%	0%	0%	**100%**	0%	0%	0%	0%	0%	0%	0%	0%	0%
Country	0%	0%	0%	0%	**96%**	4%	0%	0%	0%	0%	0%	0%	0%
Blues	0%	0%	2%	0%	5%	**81%**	0%	5%	0%	0%	0%	5%	2%
Jazz	0%	0%	0%	0%	0%	0%	**85%**	8%	0%	0%	0%	8%	0%
Soul	0%	0%	0%	0%	3%	0%	0%	**87%**	0%	0%	0%	3%	7%
Klassik	0%	0%	0%	0%	0%	0%	0%	0%	**100%**	0%	0%	0%	0%
Elektronisch	0%	0%	0%	0%	0%	0%	0%	0%	0%	**100%**	0%	0%	0%
RnB	0%	0%	0%	13%	0%	0%	0%	0%	0%	0%	**88%**	0%	0%
Folk	0%	0%	0%	0%	2%	0%	0%	0%	0%	0%	0%	**98%**	0%
Funk	0%	0%	2%	0%	0%	2%	0%	4%	0%	0%	0%	0%	**91%**

Beeindruckend ist die hohe Erkennrate bei den Genres *Reggea, Rap, Klassik* und *Elektronische Musik.*

Die Erkennraten sind generell ziemlich gut. Doch worauf achtet das Netz? Untersuchen wir für die Beantwortung dieser Frage die einzelnen Indikatornetze:

Die Indikatornetze der 1. Schicht nehmen jeweils eine Spalte eines Sonogramms entgegen. Ein Indikatornetz multipliziert jeden Helligkeitswert in dieser Spalte mit einem Gewicht. Wenn wir diese Gewichte darstellen, können wir erkennen, worauf ein Indikatornetz achtet. In der folgenden Grafik sind die Gewichte aller 50 Indikatornetze der 1. Schicht aufgeführt:

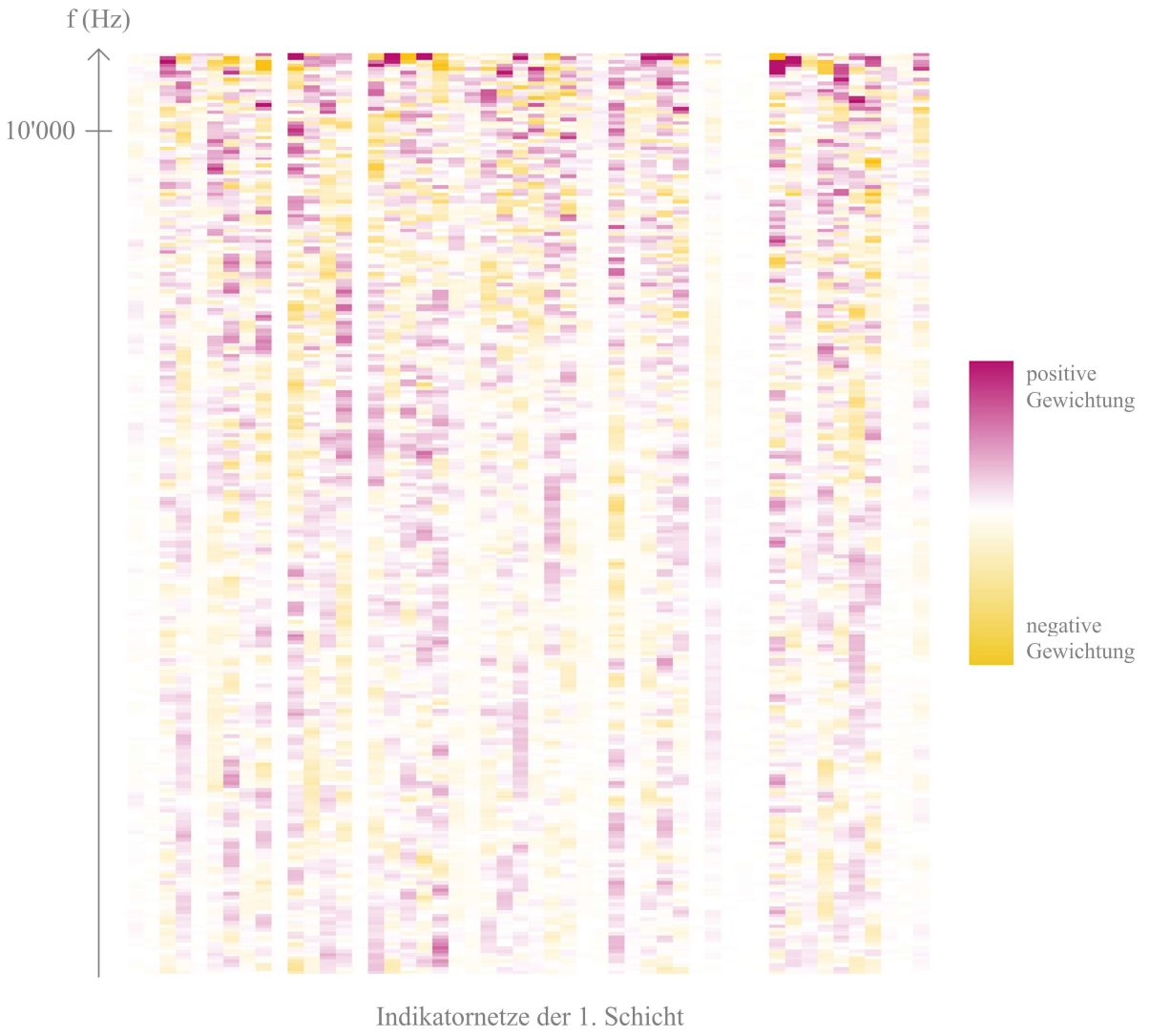


Abbildung 43: Die Gewichte aller 50 Indikatornetze der 1. Schicht.

Interessanterweise werden die höchsten Frequenzen oft am stärksten gewichtet – wir Menschen sind bei der Genreklassifikation auf diese Frequenzen nicht unbedingt angewiesen.

Auf den ersten Blick ist nicht zu erkennen, ob die einzelnen Indikatornetze wirklich für verschiedene Eigenschaften stehen. Um dies zu untersuchen, höre ich mir für jedes Indikatornetz die Songs mit den höchsten Ausgabewerten an. Für die meisten Indikatoren sind dies sehr unterschiedliche Songs. Es ist bei den meisten Indikatornetzen nicht wirklich ersichtlich, ob ein hoher Ausgabewert eine (für uns sinnvolle) Eigenschaft andeutet. Dies zeigt auf, dass wir nicht wirklich verstehen, wie das neuronales Netz Informationen aus dem Sonogramm herausfiltert.

Wie sehen die Ausgabewerte dieser 50 Indikatornetze für Songs der verschiedenen Genres aus? In der nächsten Grafik sind die durchschnittlichen Ausgabewerte der Songs, gruppiert nach Playlists, zu sehen:

Abbildung 44: Die durchschnittlichen Ausgabewerte der Indikatornetze der ersten Schicht, gruppiert nach Genre.

Die Unterschiede zwischen den einzelnen Genres sind nicht besonders gross. Lediglich klassische Musik sticht sofort hervor.

Die entsprechenden Grafiken für die Ausgabewerte der Indikatornetze der zweiten und dritten Schicht sehen ähnlich aus:

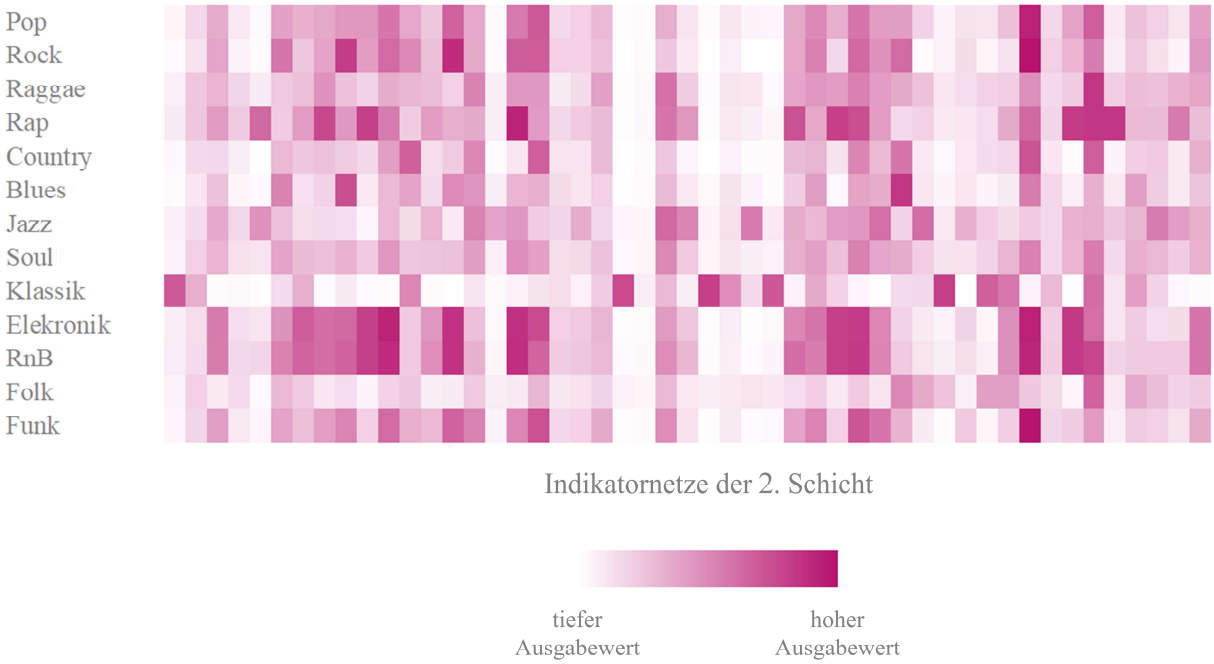


Abbildung 45: Die durchschnittlichen Ausgabewerte der Indikatornetze der zweiten Schicht, gruppiert nach Genre.

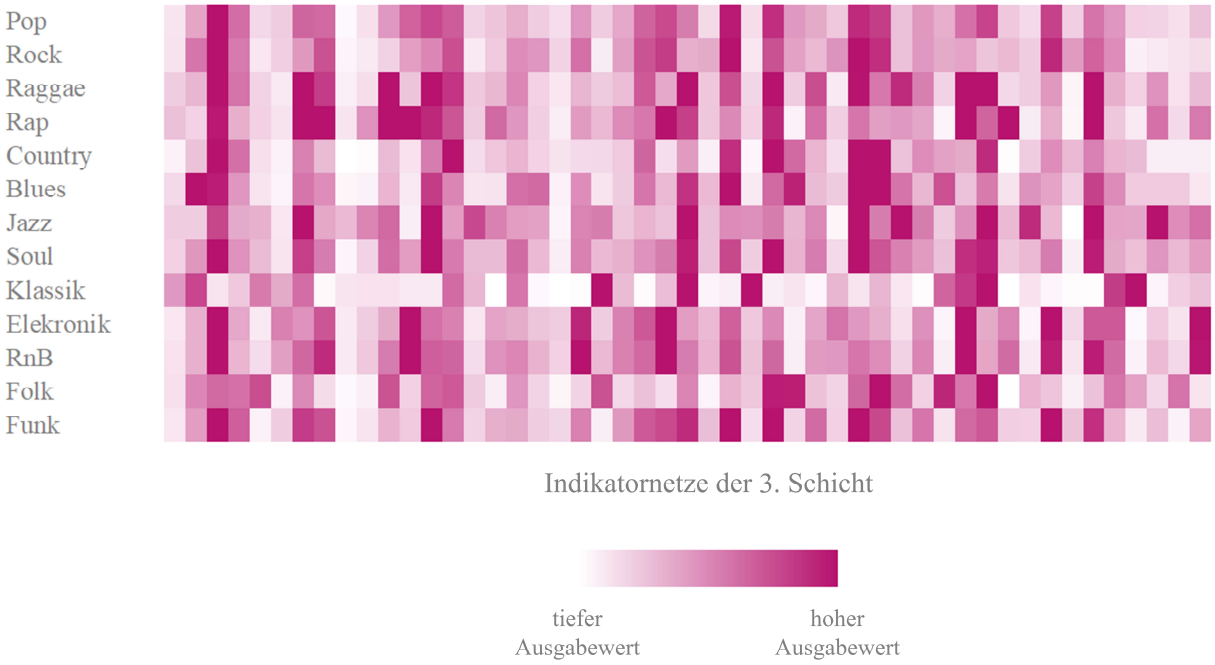


Abbildung 46: Die durchschnittlichen Ausgabewerte der Indikatornetze der dritten Schicht, gruppiert nach Genre.

5.6.1 Fazit

Es hat sich somit herausgestellt, dass das trainierte Netz tatsächlich das Genre eines Songs erkennen kann. Faszinierenderweise ist völlig unklar, warum es dazu in der Lage ist. Bei den allermeisten Indikatornetzen habe ich nicht herausfinden können, welche Eigenschaft ein hoher Ausgabewert andeuten sollte.

6 Erstellen von Playlists

Es hat sich gezeigt, dass neuronale Netze durchaus in der Lage sind, aus einem Sonogramm Informationen zu einem Song herauszulesen und mithilfe dieser Informationen das Genre des Songs relativ treffsicher zu erraten. Es stellt sich jetzt die Frage, ob es möglich ist, ein neuronales Netz zu entwerfen, mit welchem wir Playlists erstellen können. In diesem Kapitel möchte ich zuerst ergründen, was eine gute Playlist überhaupt ist. Danach beschreibe ich den Weg von der Grundidee über verschiedene Ansätze bis hin zum endgültigen Training eines Netzes und der abschliessenden Erstellung von Playlists.

6.1 Vorüberlegungen

Was zeichnet eine gute Playlist aus? Recherchiert man im Internet nach dieser Frage, tauchen einige Dinge immer wieder auf (z.B. auf [29], [30] und [31]):

- Eine Playlist sollte nur Songs beinhalten, die entweder zu einer Aktivität (Essen, Joggen, Aufräumen…) passen oder eine bestimmte Stimmung (Glücklichkeit, Entspannung, Konzentration…) auslösen.
- Die Playlist sollte mit einem sehr guten Song starten.
- Eine gute Playlist sollte auch Überraschungen enthalten, die allerdings trotzdem zum gewählten Thema passen. Dies könnte zum Beispiel ein Song aus einem anderen Jahrzehnt sein.
- Aufeinanderfolgende Songs sollten sorgfältig ausgewählt werden, sodass die Übergänge passend sind.
- Allgemein sollte mit der Reihenfolge der Songs versucht werden, Spannung aufzubauen und diese gegen Ende wieder abzubauen.
- Eine Playlist sollte bekannte und unbekannte Songs beinhalten.

Offensichtlich sind all diese Bedingungen nur mit einer sehr durchdachten Vorgehensweise erfüllbar. Um den Rahmen dieser Arbeit nicht zu sprengen, konzentriere ich mich ausschliesslich auf den ersten Aspekt: Eine Playlist definiere ich für diese Arbeit somit als eine Ansammlung von Songs, die zueinander passen und beispielsweise eine ähnliche Stimmung auslösen.

6.2 Einordnen von Musik

Um diese Playlists zu erstellen, müssen wir zuerst angeben können, ob zwei Songs zusammen in eine Playlist passen. Welche Kriterien könnten wir für die Beantwortung dieser Frage in Betracht ziehen?

Eine Antwort kann sehr unterschiedlich ausfallen. Jemand, der gerne und regelmässig Jazz hört, wird bei zwei Jazztiteln viel mehr Unterschiede und Gemeinsamkeiten erkennen und auf andere Dinge achten als jemand, für den «dies alles gleich klingt». Trotzdem können wir Songs verschiedene Merkmale zuordnen, anhand derer wir grob erkennen, ob diese zusammenpassen. Eine kleine Auswahl solcher Merkmale möchte ich nun kurz vorstellen:

6.2.1 Genre

Die Einordnung in verschiedene Genres ist wahrscheinlich die populärste Methode, Musik einzuordnen. Jeder kann in den meisten Fällen sofort bestimmen, ob er gerade einem Rocksong, einem Popsong oder einer klassischen Oper zuhört (vergl. [32]). Zudem besteht ein gewisser Konsens darüber, welche Songs welchen Genres zugeordnet werden.

Auf den ersten Blick eignen sich Genres auch, um einen Song sehr detailliert einzuordnen: Wikipedia zum Beispiel listet über 100 Genres und Subgenres auf (siehe [33]). Allerdings hat sich gezeigt, dass es oftmals sehr schwer ist, einen Song in eines der vielen Subgenres einzuordnen und dass sich sogar Experten nicht immer einig sind (vergl. [34]). Richtig zuverlässig unterteilen lassen sich Songs also trotz der grossen Anzahl Subgenres nur in sehr breite Kategorien wie Pop, Rock oder Klassik.

6.2.2 Künstler / Album

Bei der Einordnung nach Künstler oder nach Album besteht das Problem der Unsicherheit nicht. Songs lassen sich (in den meisten Fällen) eindeutig einem Album oder den Künstlern zuordnen. Allerdings sind diese zwei Informationen als Kriterien nicht immer geeignet: Ein Künstler kann im Laufe der Zeit mehrere Musikstile ausprobieren, genau wie ein Album ziemlich unterschiedliche Songs enthalten kann.

6.2.3 Stimmung

Eine interessante Einordnung ist die nach der Stimmung, welche ein Song auslöst: Wird man traurig, sobald man einen Song hört? Oder fühlt man sich versetzt in eine Bar an einem jamaikanischen Strand, in der man, einen Cocktail trinkend, alle Sorgen vergessen kann? Die Einordnung nach Stimmung ist vielleicht die, welche unserem Empfinden von Musik am nächsten kommt. Sie ist allerdings auch sehr subjektiv – vielleicht verknüpft jemand mit einem Song eine Erinnerung an eine schönere Zeit, während jemand anderes ihn einfach als alten Sommerhit abstempelt, den er sowieso schon viel zu oft gehört hat. Entsprechend schwierig gestaltet sich die Einordnung, aber gerade Spotify hat mit der Einordnung nach Stimmung sehr gute Erfahrungen gemacht (siehe [35]).

6.2.4 Fazit

Wie wir gesehen haben, gibt es nicht *die* Kriterien, mit denen wir genau angeben können, ob zwei Songs zusammen in eine Playlist passen. Das Genre und der Künstler mögen Anhaltspunkte sein – sie alleine reichen allerdings nicht aus. Die auslösende Stimmung eines Songs klingt vielversprechender – auf diese können wir hingegen nicht wirklich zugreifen.

Doch wie sich herausstellt, müssen wir solche Kriterien gar nicht kennen. Wir überlassen die Suche nach ihnen einfach neuronalen Netzen. Als Hilfe zeigen wir ihnen Playlists, die von Menschen erstellt wurden.

In den folgenden Abschnitten möchte ich Schritt für Schritt meine genaue Vorgehensweise erklären.

6.3 Datengrundlage

Zuerst sehen wir uns an, woher wir die Playlists beschaffen können, die dem neuronalen Netz als Trainingsmaterial dienen. Auf meiner Suche bin ich wieder auf die Playlists von Spotify gestossen: Es gibt Playlists mit den neusten Rap-Tracks, solche mit Klassikern aus den 80ern und solche mit Songs, die eine glückliche Stimmung auslösen sollten. Sie stellen also einen Mix der oben genannten Merkmale dar – perfekt für meine Zwecke.

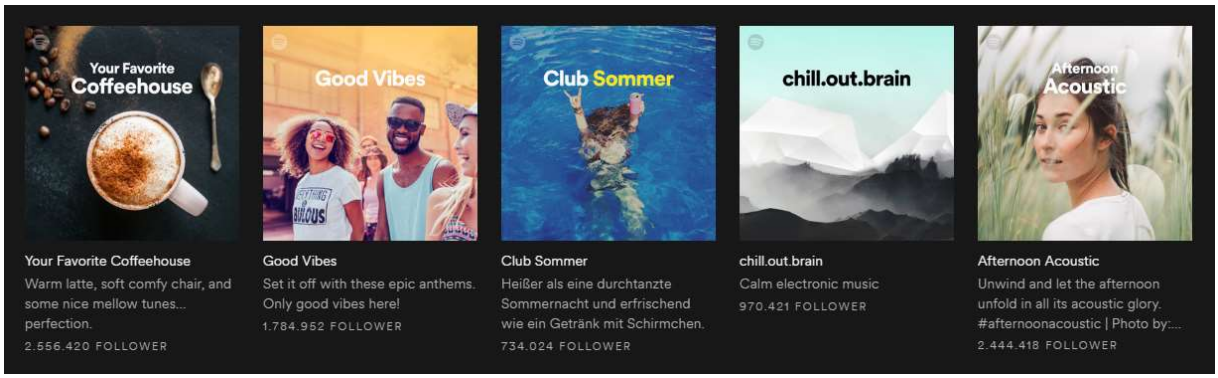


Abbildung 47: Fünf der 1'952 Playlists von Spotify, welche ich als Trainingsmaterial verwende. (Quelle: Screenshot Spotify)

Ich lade mir also die Songs aller 1'952 Playlists herunter, die Spotify selbst veröffentlicht hat. Von diesen immerhin über 55'000 Songs erstelle ich anschliessend die Sonogramme.

6.4 Grundlegender Ansatz

Wir haben bereits gesehen, wie wir neuronale Netze für Klassifikationsprobleme einsetzen. Wie können wir einem neuronalen Netz nun aber beibringen, anhand unserer Playlists für einen Song bestimmte Kriterien herauszuarbeiten? Kriterien, die wir selbst nicht kennen, anhand denen wir aber dennoch bestimmen können, ob Songs zusammen in eine Playlist passen?

Der einfachste Ansatz sieht so aus: Wir erstellen ein neuronales Netz, welches zwei Songs entgegennimmt und genau einen Wert ausgibt. Falls die beiden Songs zusammen in eine Playlist passen, sollte das Netz einen hohen Wert ausgeben. Im Training übergeben wir dem Netz abwechselnd zwei Songs aus derselben und aus unterschiedlichen Playlists. Bei Songs aus derselben Playlist geben wir als gewünschten Ausgabewert 1 an, bei Songs aus unterschiedlichen 0.

So simpel dieser Ansatz ist, so schlecht funktioniert er: Egal bei welcher Netzarchitektur, Lernrate oder bei welchem Regularisationsparameter gibt das Netz mit der Zeit für jegliche Songpaare den Wert 0.5 aus.

Ich verwende deshalb einen anderen Ansatz, welcher beispielsweise auch von [36] oder [37] eingesetzt wurde:

Wir bauen uns ein neuronales Netz, welches genau gleich aufgebaut ist wie jenes, welches wir bei der Genre-Klassifikation verwendet haben: zuerst die drei Schichten mit den je 50

Indikatornetzen und anschliessend eine normale Schicht bestehend aus 100 Neuronen. Allerdings nehmen wir statt 13 20 Ausgabeneuronen.

Die 20 Ausgabewerte repräsentieren 20 verschiedene Kriterien, anhand derer das Netz einen Song beurteilt. Unser Ziel ist, dass diese 20 Ausgabewerte andeuten, ob zwei Songs zusammen in einer Playlist sein können. In diesem Fall sollten die Kriterien für beide Songs übereinstimmen: Die 20 Ausgabeneuronen sollten ähnliche Werte ausgeben.

Mit dem Abstand der Ausgabewerte y_A und y_B von zwei Songs A und B können wir also angeben, ob das Netz vermutet, dass diese Songs zusammenpassen:

$$\sigma = \sqrt{\sum_i (y_{Ai} - y_{Bi})^2}. \tag{10}$$

Ein kleiner Abstand deutet darauf hin, dass die beiden Songs wahrscheinlich zusammen in einer Playlist sein könnten.

Unser Ziel ist es also, ein Netz so zu trainieren, dass die Ausgabewerte von Songs, die zusammen in einer Playlist sein könnten, nahe zusammenliegen. Bezogen auf die Playlists von Spotify lautet unser Ziel wie folgt: Die Ausgabewerte zweier Songs, die zusammen in einer Playlist sind, sollten näher zusammen sein als die Ausgabewerte zweier Songs, die dies nicht sind.

Allerdings habe wir neuronale Netze bisher trainiert, indem wir dem Netz Eingabewerte und die dazu passenden Ausgabewerte präsentiert und dann die Gewichte und Schwellenwerte entsprechend angepasst haben. Dies funktioniert nun nicht mehr, da wir ja nicht jedem Song gewünschte Ausgabewerte zuordnen können. Es zeigt sich allerdings, dass wir dieses Problem mit einem Trick umgehen können:

Für die Lösung müssen wir uns in Erinnerung rufen, wie wir im Training die Gewichte und Schwellenwerte anpassen: Wir verändern diese so, dass die Fehlerfunktion bei Eingabe eines Beispiels einen möglichst kleinen Wert annimmt.

Wir führen nun eine neue Fehlerfunktion ein, welche einen kleinen Wert annimmt, falls die Ausgabewerte von einem Song nahe an denen von Songs aus derselben Playlist und weit entfernt von denen von Songs anderer Playlists sind. Ich habe verschiedene Fehlerfunktionen getestet, die ich in den nächsten Abschnitten vorstellen möchte.

6.5 Getestete Fehlerfunktionen

6.5.1 Trio-Fehlerfunktion

Der erste Ansatz, den ich verfolge, stammt aus dem Paper *FaceNet: A Unified Embedding for Face Recognition and Clustering* ([37]). Die Idee ist die folgende:

Wir wählen zuerst zufällig einen Song A aus. Danach wählen wir einen Song B aus, dieser muss zusammen mit Song A in einer Playlist sein. Zuletzt wählen wir zusätzlich einen Song C aus, der sich nicht mit Song A in einer Playlist befindet. Wenn wir also

Song A unserem Netz übergeben, sollten die Ausgabewerte näher an den Ausgabewerten von Song B als an denjenigen von Song C sein.

Eine entsprechende Fehlerfunktion besteht nun aus zwei Teilen:

Der erste Teil sorgt dafür, dass die Ausgabewerte des Songs A nahe an denjenigen des Songs B liegen. Wir wissen bereits, dass wir dies mit unserer Version des euklidischen Abstandes erreichen können.

Der zweite Teil sorgt dafür, dass die Ausgabewerte des Songs A weit entfernt von denjenigen des Songs C sind. Dieser Teil der Fehlerfunktion sollte also einen kleinen Wert annehmen, falls y_A weit von y_C entfernt ist. Wir können dafür den *negativen* Abstand verwenden:

$$c(y_A, y_B, y_C) = \frac{1}{2}\sum_i \left(y_{Ai} - y_{Bi}\right)^2 - \frac{1}{2}\sum_i \left(y_{Ai} - y_{Ci}\right)^2. \tag{11}$$

6.5.2 Abgeänderte Trio-Fehlerfunktion

Als ich die Fehlerfunktion aus dem letzten Abschnitt genauer untersucht habe, ist mir ein Detail aufgefallen, welches eventuell eine Schwäche darstellen könnte. Ich schlage deshalb eine andere Trio-Fehlerfunktion vor:

$$c(y_A, y_B, y_C) = \frac{1}{2}\sum_i \left(y_{Ai} - y_{Bi}\right)^2 + \sum_i \left(\frac{y_{Ai}^2}{2} - y_{A_i} y_{C_i} - \left|y_{A_i} - y_{C_i}\right|\right). \tag{12}$$

Wie sich herausstellen wird, ist diese Fehlerfunktion (12) der Fehlerfunktion (11) in der Praxis tatsächlich überlegen.

Die genaueren Überlegungen und die Herleitung sind im Anhang zu finden.

6.5.3 Quartett-Fehlerfunktion

Nachdem ich durch die Änderung der ersten Fehlerfunktion bereits eine Verbesserung erzielt habe, überlege ich mir, ob man noch weitere Optimierungen durchführen könnte. Ich probiere einige Varianten aus, von denen vor allem eine durch ihre Resultate heraussticht:

Wir wählen zuerst zwei Playlists aus und nehmen dann aus jeder Playlist zwei Songs, Song A und Song B aus der ersten und Song C und Song D aus der zweiten Playlist.

Wir berechnen die Ausgabewerte bei Eingabe der vier Songs und ermitteln, welche zwei Songs, die nicht zusammen in einer Playlist sind, sich am nächsten sind. Wir nehmen an, dies seien im Beispiel die Songs A und C. Nun passen wir die Gewichte bei Eingabe von Song A mit der abgeänderten Fehlerfunktion an:

$$c(y_A, y_B, y_C) = \frac{1}{2}\sum_i \left(y_{Ai} - y_{Bi}\right)^2 + \sum_i \left(\frac{y_{Ai}^2}{2} - y_{A_i} y_{C_i} - \left|y_{A_i} - y_{C_i}\right|\right). \tag{13}$$

Wir sorgen dafür, dass sich die Ausgabe von Song A an die von Song B annähert und sich von der von Song C entfernt.

Zusätzlich sorgen wir mit derselben Fehlerfunktion dafür, dass sich die Ausgabe von Song C an die von Song D annähert und sich von der von Song A entfernt.

Zusammengefasst bestimmen wir zuerst die zwei Songs, die sich am nächsten sind. Danach passen wir die Gewichte so an, dass sich diese zwei Songs voneinander entfernen und sich gleichzeitig an den jeweils anderen Song der gleichen Playlist annähern.

6.6 Erste Versuche

Um die verschiedenen Fehlerfunktionen zu testen, erstelle ich ein neuronales Netz, welches weitgehend identisch aufgebaut ist wie das der Genre-Klassifikation. Nur besitzt es statt der 13 Ausgabeneuronen 20 Ausgabeneuronen. Genau wie bei der Genre-Klassifikation greife ich auch hier auf Keras zurück, da das Training sonst zu viel Zeit beanspruchen würde.

Um die Netze vergleichen zu können, nutze ich folgende Methode: Ich wähle zuerst zufällig 5'000 Songs (Song A) aus dem Testdatensatz aus. Danach wähle ich zusätzlich für jeden dieser Songs einen Song aus, der gemeinsam mit diesem in einer Playlist ist (Song B) und einen, für den dies nicht zutrifft (Song C). Nun kann ich angeben, wie gross der Anteil der Songtrios ist, bei denen der Abstand der Ausgabewerte von Song A und Song B kleiner ist als der Abstand der Ausgabewerte von Song A und Song C. Diesen Anteil, ich nenne ihn Erfolgsquote, verwende ich zum Vergleichen der Netze.

Leider muss ich schnell mit einiger Enttäuschung feststellen, dass das Training nicht erfolgreich verläuft: Das Netz erreicht zwar bereits zu Beginn mit zufälligen Gewichten und Schwellenwerten eine Erfolgsquote von 60 %. Allerdings verbessert es sich dann im Verlaufe des Trainings nicht – egal mit welcher Fehlerfunktion, Netzarchitektur oder Lernrate. Obwohl in den Sonogrammen anscheinend tatsächlich nützliche Informationen enthalten sind, kann das Netz keine sinnvollen Kriterien bilden.

Ich nehme an, dass das Hauptproblem bei den Indikatornetzen liegt: Falls es den Indikatornetzen im Training nicht gelingt, sinnvolle Gewichte zu finden, können auch die hinteren Schichten keine nützlichen Informationen ausgeben. Jetzt ist es aber so, dass wir bereits Indikatornetze besitzen, die dem Sonogramm nützliche Informationen entlocken können: die aus dem fertig trainierten Netz für die Genre-Klassifikation!

Wir können zuerst für jeden Song die Ausgabewerte der trainierten Indikatornetze berechnen und dann diese Ausgabewerte unserem Netz übergeben. Dieses Verfahren haben wir bereits unter dem Namen Pretraining und Finetuning kennengelernt.

6.7 Versuche mit Pretraining und Finetuning

6.7.1 Pretraining

Im Kapitel zur Genre-Klassifikation haben wir einem neuronalen Netz beigebracht, einen Song in eines von 13 Genres einzuordnen. Allerdings decken diese 13 Genres noch lange nicht die gesamte Musiklandschaft ab. Wie wir uns ausserdem weiter oben angesehen haben, ist für die Einordung zweier Songs auch die Stimmung, welche sie auslösen, von Bedeutung. Aus diesen Gründen füge ich zu den 13 Playlists noch 22 weitere hinzu:

Playlist	*Playlist*	*Playlist*
Classical Essentials	Fresh Electronic	Deep Sleep
Classic Pop Picks	Happy House	Hanging Out and Relaxing
Rock This	Post Metal	Songs to Sing in the Shower
Rapcaviar	Totally Alternative	Feelin Good
Are & Be	Ultimate Indie	Melancholia
Workday Soul	Latin Summer	Disco Forever
Workday Jazz	Dance Party	All Out 60s
Still Got the Blues	Seize the Day	All Out 70s
Country Icon	Instrumental Study	All Out 80s
Reggae Classics	Summer of Love	All Out 90s
Essential Folk	Young & Free	All Out 00s
All Funked Up	Workout	

Anschliessend trainiere ich ein Netz, zu erkennen, aus welcher dieser 35 Playlists ein Song stammt. Das fertig trainierte Netz ordnet einem Song in 43 % der Fälle die richtige Playlist zu.

Danach lasse ich mir für jeden der 55'000 Songs die durchschnittlichen Ausgabewerte der fertig trainierten Indikatornetze ausgeben. Anders als bei der Klassifikation verwende ich hier die durchschnittlichen Ausgabewerte der Indikatornetze aller drei Schichten. Zusätzlich berechne ich die Ausgabewerte der 100 Neuronen des neuronalen Netzes am Schluss.

Für jeden Song habe ich nun also eine Liste mit $3 \cdot 50 + 100 = 250$ Werten.

6.7.2 Finetuning

Ich erstelle jetzt ein neuronales Netz, welches diese 250 Werte entgegennimmt und aus zwei Schichten mit 100 und 20 Neuronen besteht. Da dieses Netz viel kleiner ist als dasjenige mit den Indikatornetzen, kann ich es mit meinem selbstgeschriebenen Programm trainieren und bin nicht mehr auf Keras angewiesen.

Wie vorher trainiere ich dieses Netz anhand meiner knapp 2'000 Playlists und den verschiedenen Fehlerfunktionen. Die Ergebnisse fallen nun viel erfreulicher aus: Gerade mit der Quartett-Fehlerfunktion kann ich die Resultate deutlich steigern.

Aus Zeitgründen führe ich jedes Experiment nur einmal durch. Zahlreiche Experimente zu Testzwecken zeigen aber, dass die Resultate sehr konstant sind (mit Abweichungen von etwa 0.5 %).

Fehlerfunktion	*Erfolgsquote (Trainingsbeispiele, maximal)*	*Erfolgsquote (Testbeispiele, maximal)*	*Laufzeit (für 400 Epochen)*
Trio-Fehlerfunktion	77 %	75 %	1 h 59 min 13 s
Trio-Fehlerfunktion, abgeändert	78 %	76 %	1 h 55 min 10 s
Quartett-Fehlerfunktion	84 %	82 %	2 h 15 min 20 s

Der Unterschied zwischen der Trio-Fehlerfunktion des Papers und der abgeänderten Trio-Fehlerfunktion sieht auf den ersten Blick nicht besonders gross aus. Die Werte in der Tabelle sind allerdings die maximal erreichten Erfolgsquoten während des gesamten Trainings. Die erreichten Erfolgsquoten am Ende des Trainings sehen völlig unterschiedlich aus: Nach 400 Epochen besitzt das Netz, welches ich mit der Trio-Fehlerfunktion trainiere, nur noch eine Erfolgsquote von 72 %. Die Erfolgsquote des Netzes trainiert mit der *abgeänderten* Trio-Fehlerfunktion liegt dagegen immer noch bei 76 %.

Beim Training mit der Trio-Fehlerfunktion bricht die Erfolgsquote also im Verlauf des Trainings wieder ein. Ich habe es nicht geschafft, dies zu vermeiden. Bei meiner abgeänderten Trio-Fehlerfunktion tritt dieses Problem nicht auf.

6.8 Versuche mit Autoencoder

Obwohl ich einigermassen zufrieden mit den bisherigen Resultaten bin, frage ich mich doch, ob noch eine Verbesserung möglich wäre: Gerade bei den ersten Versuchen mit der Erstellung von Playlists fallen die Ergebnisse ziemlich ernüchternd aus.

Bei meinen Überlegungen möchte ich mich vor allem auf die Funktion der 20 Ausgabeneuronen als Kriterien für die Beurteilung eines Songs konzentrieren. Ich suche nach einer Möglichkeit, dem Netz zu helfen, Ausgabewerte für einen Song auszugeben, die diesen Song möglichst gut beschreiben. Daneben sollten diese Ausgabewerte weiterhin die Eigenschaft besitzen, dass sie möglichst nahe an denjenigen von Songs aus derselben Playlist und weit entfernt von denen von Songs anderer Playlists sind.

Eine Methode, die die Ausgabewerte zwingt, die Eingabe möglichst gut zu beschreiben, ist die Verwendung eines sogenannten Autoencoders.

6.8.1 Autoencoder

Ein Autoencoder besteht aus zwei Netzen: einem Encoder-Netz und einem Decoder-Netz. Sehen wir uns seine Funktionsweise anhand der altbekannten Ziffern an (vergl. [38]):

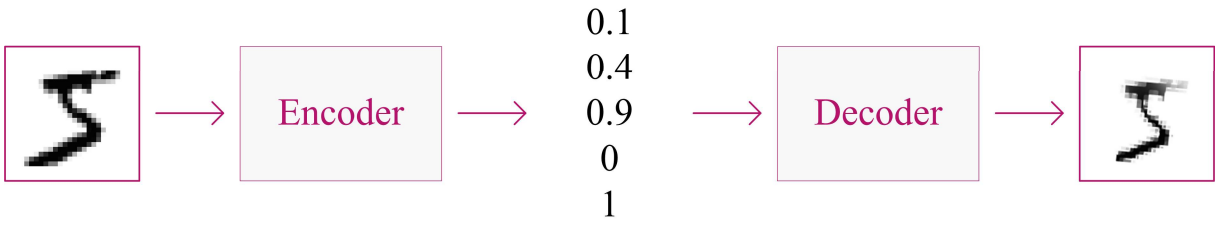


Abbildung 48: Ein Encoder besteht aus einem Encoder- und einem Decoder-Netz. Das Decoder-Netz nimmt die Ausgabewerte des Encoder-Netzes als Eingabe auf. Man trainiert den Autoencoder so, dass die Ausgabewerte des Decoders möglichst nahe an den Eingabewerten des Encoders sind.

Das Encoder-Netz nimmt als Eingabe die 784 Helligkeitswerte des Bildes entgegen und gibt eine kleine Anzahl von Werten aus, beispielsweise fünf. Diese fünf Ausgabewerte übergeben wir dem Decoder-Netz als Eingabe, welches 784 Ausgabewerte ausgibt. Wir trainieren diese zwei Netze nun so, dass die Ausgabewerte des Decoder-Netzes möglichst nahe an den Eingabewerten des Encoder-Netzes, den Helligkeitswerten des Bildes, liegen. Doch worin liegt der Sinn dieser Methode?

Der Schlüssel liegt in den fünf Ausgabewerten des Encoders. Diese fünf Werte müssen genügend Informationen über das Bild enthalten, sodass es möglichst gut durch den Decoder wiederhergestellt werden kann. Das Encoder-Netz liefert sozusagen die Anleitung, um das Bild zu zeichnen – und das Decoder-Netz setzt diese Anleitung wieder in ein Bild um.

Ein Autoencoder wird mit folgender Fehlerfunktion trainiert:

$$c(x, y_{Decoder}) = \frac{1}{2}\sum_i \left(y_{Decoder\,i} - x_i\right)^2. \tag{14}$$

Der Fehler misst den Abstand zwischen den Eingabewerten x und den Ausgabewerten des Decoders $y_{Decoder}$.

6.8.2 Grundidee

Ich hoffe, dass bei einer Kombination von Autoencoder und Quartett-Fehlerfunktion der Encoder bei Eingabe eines Songs Werte ausgibt, die zwei Bedingungen erfüllen: Zum einen sollten sie den Song möglichst gut beschreiben. Zum anderen sollten sie möglichst nahe an den Ausgabewerten von Songs aus derselben Playlist und weit entfernt von den Ausgabewerten von Songs anderer Playlists sein.

Wir könnten nun den Encoder mit Indikatornetzen ausstatten und diesem die Helligkeitswerte eines Sonogramms übergeben. Spätestens wenn wir uns aber überlegen, was die Aufgabe des Decoders dann wäre, merken wir, dass diese Strategie nicht funktionieren kann: Auch wenn wir den gewählten Songausschnitt ein wenig verschieben würden, sollte der Encoder in beiden Fällen ähnliche Werte ausgeben, was er dank der Indikatornetze auch machen wird. Der Decoder müsste in den beiden Fällen aber ziemlich unterschiedliche Ausgabewerte ausgeben – nämlich die Helligkeitswerte der verschobenen Sonogramme.

Aus diesem Grund müssen wir auch hier auf unsere 250 Ausgabewerte der vortrainierten Netze zurückgreifen und diese dem Encoder übergeben. Diese Werte sind auch die gewünschten Ausgabewerte des Decoders.

Wenn wir nur einen klassischen Autoencoder ohne Modifikationen einsetzen, können wir noch kein besseres Resultat erzielen als mit der Quartett-Fehlerfunktion: Die Erfolgsquote liegt bei 78 %.

Wir können jetzt die Quartett-Fehlerfunktion und den Autoencoder kombinieren. Dazu gehen wir folgendermassen vor:

6.8.3 Umsetzung

Wir wählen zuerst die vier Songs unter Berücksichtigung der bekannten Bedingungen aus. Die 250 Eingabewerte nennen wir x_A, x_B, x_C und x_D. Wir berechnen die Ausgabewerte des Encoders und Decoders für alle vier Songs. Die Ausgabewerte des Encoders nennen wir y_A, y_B, y_C und y_D, die Ausgabewerte des Decoders z_A, z_B, z_C und z_D.

Danach bestimmen wir, welche beiden Songs, die nicht in einer gemeinsamen Playlist sind, die nächsten Ausgabewerte besitzen. Seien dies im Beispiel Song A und Song C.

Anschliessend passen wir die Gewichte und Schwellenwerte nach Eingabe von Song A mit folgender Fehlerfunktion an:

$$c(y_A, y_B, y_C) = \frac{1}{2}\sum_i \left(y_{Ai} - \mathrm{y}_{Ci}\right)^2 + \sum_i \left(\frac{{y_{Ai}}^2}{2} - y_{A_i}\mathrm{y}_{C_i} - \left|y_{A_i} - y_{C_i}\right|\right) + q \cdot \frac{1}{2}\sum_i \left(z_{Ai} - x_{Ai}\right)^2. \tag{15}$$

Diese Fehlerfunktion ist eine Kombination aus der Quartett-Fehlerfunktion für Song A (12) und der Fehlerfunktion für den Autoencoder (14). q bestimmt, wie stark die Fehlerfunktion des Autoencoders gewichtet wird. Im Fall von $q = 0$ verhält sich (15) wie die normale Quartett-Fehlerfunktion. Als optimal hat sich ein Faktor von $q = 1.5$ herausgestellt.

Für Song C lautet die Fehlerfunktion analog:

$$c(y_C, y_D, y_A) = \frac{1}{2}\sum_i \left(y_{Ci} - \mathrm{y}_{Ai}\right)^2 + \sum_i \left(\frac{{y_{Ci}}^2}{2} - y_{C_i}\mathrm{y}_{A_i} - \left|y_{C_i} - y_{A_i}\right|\right) + q \cdot \frac{1}{2}\sum_i \left(z_{Ci} - x_{Ci}\right)^2. \tag{16}$$

Zum Schluss trainieren wir den Autoencoder bei Eingabe der Songs B und D:

$$c(y_B) = q \cdot \frac{1}{2}\sum_i \left(z_{Bi} - x_{Bi}\right)^2. \tag{17}$$

$$c(y_B) = q \cdot \frac{1}{2}\sum_i \left(z_{Di} - x_{Di}\right)^2. \tag{18}$$

6.8.4 Cross-Playlist-Encoder

Für eine letzte Modifikation des Autoencoders wurde ich inspiriert durch das bereits erwähnte Paper *Colorful Image Colorization* von Richard Zhang, Phillip Isola und Alexei A. Efros (siehe [3]). In diesem benutzten die Autoren unter anderem einen Autoencoder, den sie mit Bildern trainierten. Anstatt als Eingabewerte und gewünschte Ausgabewerte einfach die Helligkeitswerte der Pixel zu nehmen, übergaben sie dem Encoder die Werte eines der drei Farbkanäle. Als gewünschte Ausgabewerte des Decoders verwendeten sie dann die Werte eines anderen Farbkanals. Diesem Autoencoder gaben sie den Namen *Cross-Channel-Encoder*:

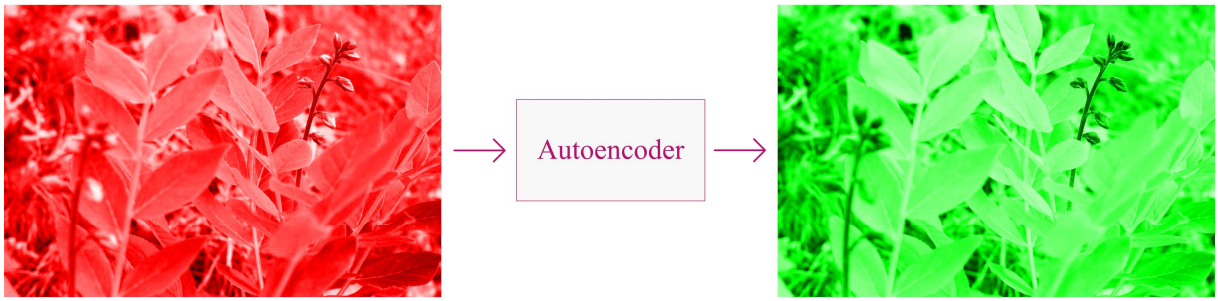


Abbildung 49: Die Autoren des Papers übergaben einem Autoencoder die Werte des roten Farbkanals. Als Ausgabe sollte der Decoder die Werte des grünen Farbkanals ausgeben.

Meine darauf basierende Idee lautet wie folgt: Ich übergebe dem Autoencoder als Eingabewerte die 250 Werte eines Songs A. Als gewünschte Ausgabewerte des Decoders gebe ich nun nicht aber diese 250 Werte an, sondern die 250 Werte eines anderen Songs B aus derselben Playlist. Der Autoencoder sollte dann folglich nicht nur lernen, den Song A zu beschreiben. Er sollte lernen, einen Song zu beschreiben, der zusammen mit Song A in einer Playlist sein könnte. Und genau dies ist ja sowieso unser Ziel.

Die Fehlerfunktion (17) passen wir beispielsweise wie folgt an:

$$c(y_B, y_A) = q \cdot \frac{1}{2} \sum_i \left(z_{Bi} - x_{Ai}\right)^2. \tag{19}$$

Wie bei den Resultaten ersichtlich sein wird, führt gerade diese Änderung zu einem deutlichen Anstieg der Erfolgsquote.

6.9 Ensembling

Zusätzlich verwende ich noch die Methode des Ensembling. Ich trainiere dafür zuerst 18 Netze mit bekanntem Aufbau in der Klassifikation der 35 Genres. Nun trainiere ich 18 Cross-Playlist-Encoder zusammen mit der Quartettfehlerfunktion, wobei ich jeden Encoder mit einem anderen Netz der Genreklassifikation vortrainiere.

Am Schluss besitze ich nun also 18 Netze, welche jeweils 20 Werte ausgeben. Die Distanz σ berechne ich nun über alle $18 \cdot 20$ Werte:

$$\sigma = \sqrt{\sum_{k=1}^{18} \sum_{i=1}^{20} \left(y_{A_{k_i}} - y_{B_{k_i}} \right)^2}. \tag{20}$$

6.10 Resultate

In der folgenden Tabelle sind die erreichten Erfolgsquoten mit den verschiedenen Ansätzen zu sehen:

Methode	*Erfolgsquote (Trainingsbeispiele, maximal)*	*Erfolgsquote (Testbeispiele, maximal)*	*Laufzeit (für 400 Epochen)*
Trio-Fehlerfunktion	77 %	75 %	1 h 59 min 13 s
Trio-Fehlerfunktion, abgeändert	78 %	76 %	1 h 55 min 10 s
Quartett-Fehlerfunktion	84 %	82 %	2 h 15 min 20 s
Autoencoder	81 %	78 %	2 h 40 min 10 s
Autoencoder mit Quartett-Fehlerfunktion	82 %	79 %	
Cross-Playlist-Encoder	84 %	82 %	2 h 42 min 35 s
Cross-Playlist-Encoder mit Quartett-Fehlerfunktion	88 %	84 %	2 h 49 min 48 s
Cross-Playlist-Encoder mit Quartett-Fehlerfunktion: Ensemble aus 18 Netzen	90 %	88 %	47 h 20 min 20 s

Der Cross-Playlist-Encoder in Verbindung mit der Quartett-Fehlerfunktion stellt sich tatsächlich als klarer Gewinner bei den einzelnen Netzen heraus. Im Vergleich zur Trio-Fehlerfunktion des Papers habe ich damit die Erfolgsquote sowohl bei den Trainings- als auch bei den Testbeispielen um ungefähr 10 % steigern können.

Das Ensemble kann die Erfolgsquote auf 90 % steigern. Dies wirkt im Vergleich zur Laufzeit auf den ersten Blick nicht besonders beeindruckend. Allerdings wird das Ensebmle sich bei der Playlisterstellung als klar überlegen herausstellen.

6.11 Analyse

In diesem Abschnitt untersuche ich, wie die Ausgabewerte eines mit dem Cross-Playlist-Encoder mit Quartett-Fehlerfunktion trainierten Netzes aussehen.

Wir können für diesen Zweck eine ähnliche Grafik wie bei der Analyse des Netzes zur Genre-Klassifikation einsetzen:

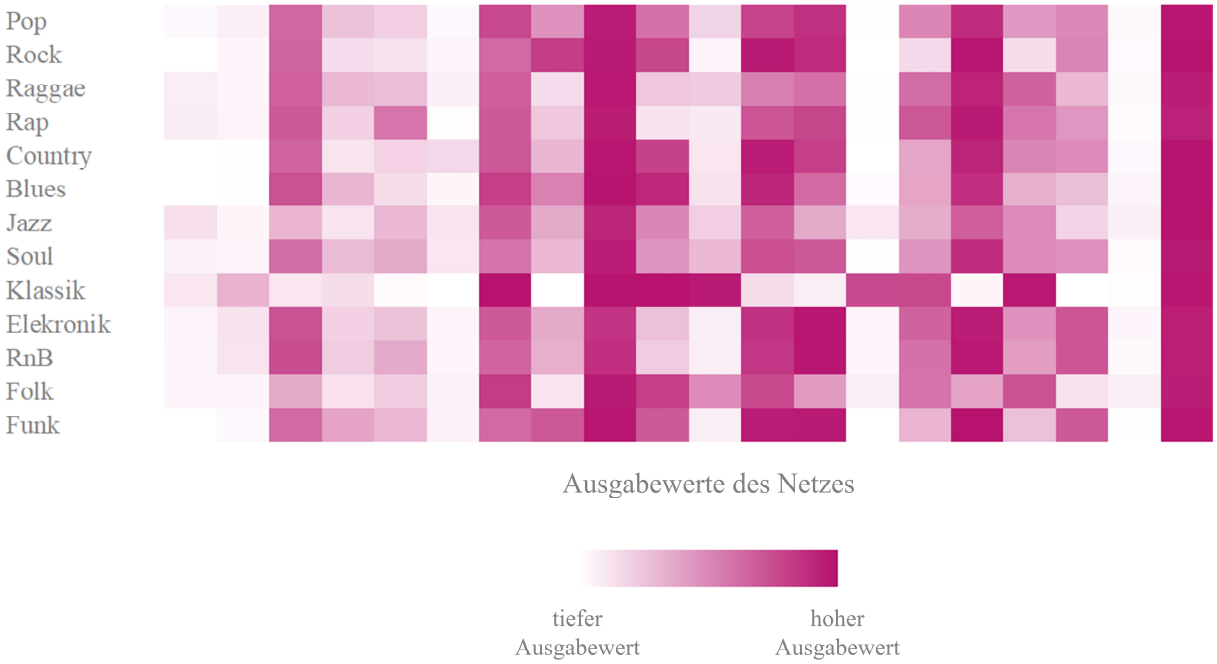


Abbildung 50: Die durchschnittlichen Ausgabewerte des Netzes, gruppiert nach Genre.

Auch hier sind sich die Ausgabewerte auf den ersten Blick relativ ähnlich. Doch wie weit auseinander sind die Ausgabewerte für einzelne Songs? Für die Beantwortung dieser Frage generiere ich 10'000 zufällige Songtrios. Die Abstände der Ausgabewerte von den Songs, die zusammen in einer Playlist sind, sind wie folgt verteilt:

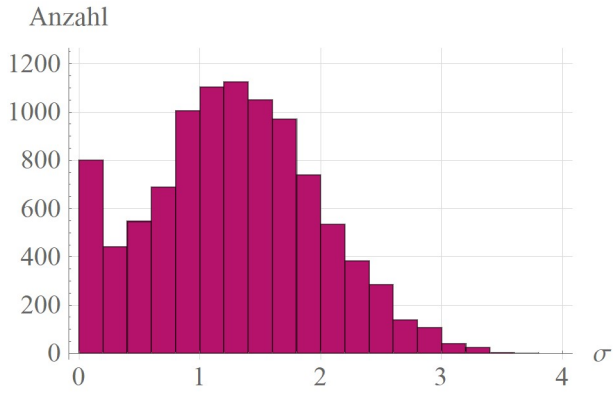


Abbildung 51: Die Verteilung der Abstände der Ausgabewerte zweier Songs derselben Playlist.

Der mittlere Abstand beträgt 1.3.

Die Abstände der Ausgabewerte von denjenigen Songs, die nicht zusammen in einer Playlist sind, sind hingegen so verteilt:

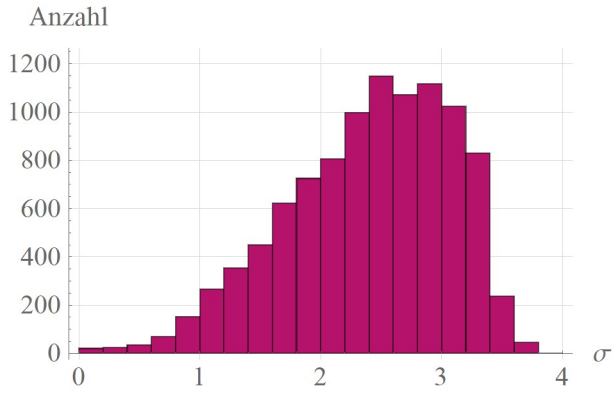


Abbildung 52: Die Verteilung der Abstände der Ausgabewerte zweier beliebiger Songs.

Der mittlere Abstand beträgt hier 2.4.

Die Kullbert-Leibniz-Distanz der beiden Verteilungen ist $KL(\sigma_{beliebig}, \sigma_{passend}) = 0.4471$.

Die unterschiedlichen Mittelwerte und Peaks zeigen schön, dass das Netz tatsächlich geeignete Kriterien bilden konnte. Genau wie bei den Indikatornetzen der Genre-Klassifikation kann ich aber leider auch hier nicht bestimmen, was die einzelnen Ausgabewerte genau bedeuten.

6.12 Erstellen von Playlists

Ich bin nun somit in der Lage, für zwei Songs bestimmen zu können, wie gut diese zusammen in eine Playlist passen. Ich möchte abschliessend versuchen, anhand gegebener Songs eine Playlist mit passenden Songs zu erstellen.

Mein Ziel lautet wie folgt: Ich möchte anhand *n* gegebener Songs als Referenz eine Playlist erstellen, die *m* Songs enthält, welche zu den Referenzsongs möglichst gut passen.

Ich beschränke mich dabei, um den Rahmen dieser Arbeit nicht zu sprengen, nur auf die offensichtlichste Methode: Wir ermitteln die Songs, deren Ausgabewerte des Ensembles am nächsten zu denjenigen der Referenzsongs sind. Wir suchen also die *m* Songs, die den kleinsten durchschnittlichen Abstand zu allen Referenzsongs besitzen.

Dieser Ansatz funktioniert vor allem, wenn die Referenzsongs nicht zu unterschiedlich sind. Die Qualität der erstellten Playlists sind meiner Meinung nach relativ unterschiedlich. In etwa ¾ der Fälle sind wirklich passende Songs unter den Vorschlägen zu finden. In einigen Fällen ist die Ähnlichkeit der vorgeschlagenen Songs mit den Referenzsongs aber nicht zu erkennen.

Hier stellt sich der Vorteil des Ensembles heraus: Bei der subjektiven Qualität der Songvorschläge ist das Ensemble einem einzelnen Netz klar überlegen. Dies hat mich überrascht, da der Unterschied der Erfolgsquoten (90 % vs. 88 %) nicht auf einen bedeutenden Leistungsunterschied hinweist.

6.13 Weiterführende Überlegungen

Ich bin soweit zufrieden mit dem Resultat: Die erzeugten Playlists enthalten teilweise wirklich passende Songs. Nichtsdestotrotz besitzt mein Ansatz einige Schwächen:

- Ich besitze nur 30-sekündige Ausschnitte jedes Songs. Wie das neuronale Netz einen Song beurteilt, hängt von dem gewählten Ausschnitt ab. Dies ist bei den meisten Songs kein Problem. Bei einigen Songs, die beispielsweise langsam beginnen und sich dann gegen Schluss steigern, kann es vorkommen, dass vielleicht nur langsame Songs vorgeschlagen werden.
- Meine Datenbasis von knapp 55'000 Songs ist relativ klein, zudem auf dem Stand von September 2017. Dies führt dazu, dass längst nicht alle Songs bei der Playlisterstellung in Betracht gezogen werden können.
- Der Einsatz von Pretraining und Finetuning hat gut funktioniert. Er bedeutet allerdings, dass das Netz für die Playlisterstellung nur so gut sein kann wie das Netz für die Genre-Klassifikation. Da bei der Genre-Klassifikation ein Song nur grob eingeordnet werden muss, kann das Netz für die Playlisterstellung gar nicht auf sehr detaillierte Informationen zurückgreifen. Dieses Problem lässt sich mit dem Ensemble zumindest teilweise lösen.
- Die abgeänderte Trio-Fehlerfunktion ist (zumindest bei dieser Aufgabe) der Trio-Fehlerfunktion des Papers überlegen. Sie funktioniert aber nur, sobald zumindest die Aktivierungsfunktion der Neuronen der letzten Schicht einen Wert zwischen 0 und 1 ausgibt. Dies ist beispielsweise bei Verwendung der Sigmoid-Aktivierungsfunktion der Fall.
- Die Quartett-Fehlerfunktion hat sich als Erfolg herausgestellt. Über die genauen Gründe für ihre Überlegenheit gegenüber einer Trio-Fehlerfunktion kann ich allerdings nur spekulieren.

Ich werde voraussichtlich auch nach Abgabe dieser Arbeit versuchen, das Resultat noch weiter zu verbessern. Ich bin überzeugt, dass das Potenzial von neuronalen Netzen bei der Erstellung von Playlists noch lange nicht ausgeschöpft ist.

Mögliche Ausbaumöglichkeiten wären beispielsweise:

- Das Pretraining könnte vielleicht durch mehr und grössere Playlists optimiert werden. Ebenfalls denkbar wäre, mehr Indikatornetze einzusetzen oder die Indikatorschichten einzeln zu trainieren.
- Wir haben gesehen, dass in der Wahl der Fehlerfunktion viel Potenzial existiert. Eventuell gibt es eine Fehlerfunktion, die für diese Aufgabe noch besser geeignet wäre.
- Das Erstellen der Playlists selbst kann noch verbessert werden: Man könnte den Ansatz, welche Songs man auswählt, sicherlich noch stark weiterentwickeln. Ausserdem könnten mehr Informationen wir zum Beispiel Popularität oder Erscheinungsjahr miteinbezogen werden.
- Vielversprechend sieht der Ansatz aus, die Anzahl der Songs für das Finetuning zu verkleinern und eine Überanpassung zu forcieren. Dies wäre in diesem Fall von Vorteil, da wir bei der Playlisterstellung sowieso nur auf diese Songs zugreifen.

- Es stellt sich die Frage, ob ein grösseres Ensemble oder weiterführende Ensemblemethoden wie das Stacking das Resultat verbessern können.

Schlussendlich wäre es natürlich interessant, eine App zu entwickeln, mit der jeder seine eigenen Playlists erstellen lassen kann. Zurzeit erstelle ich für mich Playlists mit einer selbstgeschriebenen App für Windows. Ich bin durch die erstellten Playlists bereits auf einige neue Songs gestossen, die mir gefallen. Eine solche App könnte also vielleicht nützlich für viele Personen sein.

7 Danksagung

Zuerst möchte ich mich ganz herzlich bei meinem Betreuer, Ueli Manz, bedanken. Er stand mir mit Rat und Tat zur Seite, brachte hilfreiche und spannende Anregungen und hat mir mit seiner konstruktiven Kritik immer weitergeholfen. Ich habe die Zusammenarbeit und Diskussionen mit ihm sehr geschätzt.

Ausserdem möchte ich mich bei meiner gesamten Familie bedanken, die mich die ganze Zeit unterstützt hat – sei es mit Ratschlägen, mit Durchlesen oder mit Ignorieren des doch recht lauten Lüftergeräusches meines Computers während der unzähligen Stunden, in denen ich Experimente durchgeführt habe.

8 Anhang

8.1 Ändern der Gewichte und Schwellenwerte

Auf die Frage, wie die Gewichte und Schwellenwerte im Training geändert werden, gehe ich im bisherigen Text nur sehr kurz ein. In diesem Abschnitt möchte ich das dazu verwendete Backpropagation-Verfahren genauer vorstellen (siehe [40]).

8.1.1 Notation

Zuerst möchte ich die bisher verwendeten Notationen erweitern und leicht abändern:

Ein Netz besteht aus n Schichten. Die k-te Schicht besitzt $n^{(k)}$ Neuronen.

Sei $N_j^{(k)}$ das j-te Neuron der k-ten Schicht:

- Seinen Ausgabewert nennen wir $x_j^{(k)}$.
- Es nimmt als Eingabewerte die Ausgabewerte der Neuronen der vorgehenden Schicht entgegen: $x_1^{(k-1)}, x_2^{(k-1)}, x_3^{(k-1)}, \dots$
- Diese Eingabewerte werden gewichtet: Der Eingabewert $x_i^{(k-1)}$ (vom i-ten Neuron der vorgehenden Schicht) wird mit dem Gewicht $w_{ij}^{(k)}$ multipliziert. Diese gewichteten Ausgabewerte werden anschliessend aufsummiert:
$$s_j^{(k)} = \sum_i x_i^{(k-1)} \cdot w_{ij}^{(k)}.$$
Auf den Schwellenwert können wir verzichten, da wir uns diesen als weiteres Neuron mit festem Ausgabewert 1 vorstellen können.
- Auf dieses Ergebnis wird schliesslich die Aktivierungsfunktion $\mathrm{a}(s_j^{(k)})$ angewendet:
$$x_j^{(k)} = \mathrm{a}\left(s_j^{(k)}\right) = a\left(\sum_i x_i^{(k-1)} \cdot w_{ij}^{(k)}\right).$$

Die Ausgabewerte des Netzes sind somit $x_1^{(n)}, x_2^{(n)}, x_3^{(n)}, \dots$

Die Eingabewerte des Netzes nennen wir $x_1^{(0)}, x_2^{(0)}, x_3^{(0)}, \dots$

8.1.1.1 Fehlerfunktion

Als Fehlerfunktion verwenden wir zur Demonstration unsere Version des euklidischen Abstandes:

$$\mathrm{c}\left(x^{(n)}, t\right) = \frac{1}{2}\sum_i \left(x_i^{(n)} - t_i\right)^2.$$

Allerdings funktioniert das Backpropagation-Verfahren mit beliebigen Fehlerfunktionen.

8.1.2 Grundsätzliches Vorgehen

Für die Änderung $\Delta w_{ij}^{(k)}$ gilt:

$$\Delta w_{ij}^{(k)} = -\alpha \cdot \frac{\partial c}{\partial w_{ij}^{(k)}}.$$

Wir müssen somit die Ableitungen des Fehlers nach allen Gewichten berechnen. Wir starten dabei am Ende des Netzes und bewegen uns schrittweise nach vorne. Von daher stammt auch der englische Name *Backpropagation*, was so viel wie *sich nach hinten ausbreiten* bedeutet. Beginnen wir mit den Gewichten der letzten Schicht.

8.1.3 Berechnen der Ableitung nach den Gewichten der letzten Schicht

Wie hängt die Fehlerfunktion von den Gewichten der letzten Schicht ab? Zunächst hängt sie von den Ausgabewerten $x_1^{(n)}, x_2^{(n)}, x_3^{(n)}, \dots$ der Neuronen der letzten Schicht ab. Diese wiederum hängen von den gewichteten Summen $s_1^{(n)}, s_2^{(n)}, s_3^{(n)}, \dots$ ab. Diese Summen hängen von den Gewichten der Verbindungen zwischen den zwei letzten Schichten ab: $w_{11}^{(n)}, w_{12}^{(n)}, w_{13}^{(n)}, \dots, w_{21}^{(n)}, w_{22}^{(n)}, w_{23}^{(n)}, \dots$

Die Ableitung der Fehlerfunktion nach den Gewichten der letzten Schicht erhalten wir nach der Kettenregel also folgendermassen:

$$\frac{\partial c}{\partial w_{ij}^{(n)}} = \frac{\partial c}{\partial x_j^{(n)}} \frac{\partial x_j^{(n)}}{\partial w_{ij}^{(n)}} = \frac{\partial c}{\partial x_j^{(n)}} \frac{\partial x_j^{(n)}}{\partial s_j^{(n)}} \frac{\partial s_j^{(n)}}{\partial w_{ij}^{(n)}}.$$

Wie können wir die einzelnen Faktoren berechnen?

- Die Ableitung $\frac{\partial c}{\partial x_j^{(n)}}$ des Fehlers nach einem Ausgabewert:

  $$\frac{\partial c}{\partial x_j^{(n)}} = \frac{\partial}{\partial x_j^{(n)}} \frac{1}{2} \sum_i \left(x_i^{(n)} - t_i \right)^2.$$

  Dank der Summenregel können wir die Summe und den Faktor vorklammern:

  $$\frac{\partial c}{\partial x_j^{(n)}} = \frac{\partial}{\partial x_j^{(n)}} \frac{1}{2} \sum_i \left(x_i^{(n)} - t_i \right)^2 = \frac{1}{2} \sum_i \frac{\partial}{\partial x_j^{(n)}} \left(x_i^{(n)} - t_i \right)^2.$$

  Von den Summanden hängt nur derjenige für $i = j$ von $x_j^{(n)}$ ab. Die Ableitung für alle anderen Summanden ist somit gleich 0:

  $$\frac{\partial c}{\partial x_j^{(n)}} = \frac{1}{2} \sum_i \frac{\partial}{\partial x_j^{(n)}} \left(x_i^{(n)} - t_i \right)^2 = \frac{1}{2} \frac{\partial}{\partial x_j^{(n)}} \left(x_j^{(n)} - t_j \right)^2.$$

  Mit der Kettenregel erhalten wir:

  $$\frac{\partial c}{\partial x_j^{(n)}} = \frac{1}{2} \frac{\partial}{\partial x_j^{(n)}} \left(x_j^{(n)} - t_j \right)^2 = \frac{1}{2} \cdot 2 \cdot \left(x_j^{(n)} - t_j \right) \cdot 1 = x_j^{(n)} - t_j.$$

  Hier zeigt sich der Grund für den Vorfaktor $\frac{1}{2}$.

- Die Ableitung $\frac{\partial x_j^{(n)}}{\partial s_j^{(n)}}$ eines Ausgabewertes nach der gewichteten Summe:

$$\frac{\partial x_j^{(n)}}{\partial s_j^{(n)}} = \frac{\partial}{\partial s_j^{(n)}} a\left(s_j^{(n)}\right) = a'\left(s_j^{(n)}\right).$$

- Die Ableitung $\frac{\partial s_j^{(n)}}{\partial w_{ij}^{(n)}}$ der gewichteten Summe nach einem Gewicht:

$$\frac{\partial s_j^{(n)}}{\partial w_{ij}^{(n)}} = x_i^{(n-1)}.$$

Zusammengefasst können wir die Ableitung eines Gewichts der letzten Schicht nach dem Fehler so berechnen:

$$\frac{\partial c}{\partial w_{ij}^{(n)}} = \frac{\partial c}{\partial x_j^{(n)}} \frac{\partial x_j^{(n)}}{\partial s_j^{(n)}} \frac{\partial s_j^{(n)}}{\partial w_{ij}^{(n)}} = \left(x_j^{(n)} - t_j\right) \cdot a'\left(s_j^{(n)}\right) \cdot x_i^{(n-1)}.$$

Wir führen nun die Bezeichnung $\delta_j^{(k)}$ ein, die wir wie folgt definieren:

$$\delta_j^{(k)} = \frac{\partial c}{\partial s_j^{(k)}}.$$

Wir können die Ableitung eines Gewichts der letzten Schicht nach dem Fehler somit auch so schreiben:

$$\frac{\partial c}{\partial w_{ij}^{(n)}} = \frac{\partial c}{\partial x_j^{(n)}} \frac{\partial x_j^{(n)}}{\partial s_j^{(n)}} \frac{\partial s_j^{(n)}}{\partial w_{ij}^{(n)}} = \delta_j^{(n)} \cdot x_i^{(n-1)}.$$

Wobei $\delta_j^{(n)}$ den folgenden Wert besitzt:

$$\delta_j^{(n)} = \frac{\partial c}{\partial s_j^{(n)}} = \left(x_j^{(n)} - t_j\right) \cdot a'\left(s_j^{(n)}\right).$$

8.1.4 Berechnen der Ableitung nach den Gewichten der inneren Schichten

Wie hängt die Fehlerfunktion von den Gewichten der zweitletzten Schicht ab? Zunächst hängt sie von den Ausgabewerten $x_1^{(n-1)}, x_2^{(n-1)}, x_3^{(n-1)}, \ldots$ der Neuronen der zweitletzten Schicht ab. Diese wiederum hängen von den gewichteten Summen $s_1^{(n-1)}, s_2^{(n-1)}, s_3^{(n-1)}, \ldots$ und diese von den Gewichten $w_{11}^{(n-1)}, w_{12}^{(n-1)}, w_{13}^{(n-1)}, \ldots, w_{21}^{(n-1)}, w_{22}^{(n-1)}, w_{23}^{(n-1)}, \ldots$ ab:

$$\frac{\partial c}{\partial w_{ij}^{(n-1)}} = \frac{\partial c}{\partial x_j^{(n-1)}} \frac{\partial x_j^{(n-1)}}{\partial s_j^{(n-1)}} \frac{\partial s_j^{(n-1)}}{\partial w_{ij}^{(n-1)}}.$$

- Die Ableitung $\frac{\partial c}{\partial x_j^{(n-1)}}$ des Fehlers nach dem Ausgabewert:

Der Ausgabewert $x_j^{(n-1)}$ beeinflusst alle gewichteten Summen der Neuronen der letzten Schicht:

$$\frac{\partial c}{\partial x_j^{(n-1)}} = \sum_l \frac{\partial c}{\partial s_l^{(n)}} \frac{\partial s_l^{(n)}}{\partial x_j^{(n-1)}}.$$

$\frac{\partial c}{\partial s_l^{(n)}}$ können wir mit $\delta_l^{(n)}$ abkürzen:

$$\frac{\partial c}{\partial x_j^{(n-1)}} = \sum_l \delta_l^{(n)} \frac{\partial s_l^{(n)}}{\partial x_j^{(n-1)}}.$$

Weiterhin folgt:

$$\frac{\partial c}{\partial x_j^{(n-1)}} = \sum_l \delta_l^{(n)} \frac{\partial s_l^{(n)}}{\partial x_j^{(n-1)}} = \sum_l \delta_l^{(n)} w_{jl}^{(n)}.$$

- Die Ableitung $\frac{\partial x_j^{(n-1)}}{\partial s_j^{(n-1)}}$ des Ausgabewertes nach der gewichteten Summe:

$$\frac{\partial x_j^{(n-1)}}{\partial s_j^{(n-1)}} = a'\left(s_j^{(n-1)}\right).$$

- Die Ableitung $\frac{\partial s_j^{(n-1)}}{\partial w_{ij}^{(n-1)}}$ der gewichteten Summe nach einem Gewicht:

$$\frac{\partial s_j^{(n-1)}}{\partial w_{ij}^{(n-1)}} = x_i^{(n-2)}.$$

Zusammengefasst folgt:

$$\frac{\partial c}{\partial w_{ij}^{(n-1)}} = \frac{\partial c}{\partial x_j^{(n-1)}} \frac{\partial x_j^{(n-1)}}{\partial s_j^{(n-1)}} \frac{\partial s_j^{(n-1)}}{\partial w_{ij}^{(n-1)}} = \sum_l (\delta_l^{(n)} w_{jl}^{(n)}) \cdot a'\left(s_j^{(n-1)}\right) \cdot x_i^{(n-2)}.$$

Wie wir sehen, müssen wir für das Berechnen der Ableitungen nach den Gewichten der zweiten Schicht nicht nochmals von vorne beginnen, sondern können mit den δ's der letzten Schicht weiterrechnen.

Für die zweitletzte Schicht lautet das $\delta_j^{(n-1)}$ wie folgt:

$$\delta_j^{(n-1)} = \frac{\partial c}{\partial s_j^{(n-1)}} = \frac{\partial c}{\partial x_j^{(n-1)}} \frac{\partial x_j^{(n-1)}}{\partial s_j^{(n-1)}} = \sum_l (\delta_l^{(n)} w_{jl}^{(n)}) \cdot a'\left(s_j^{(n-1)}\right).$$

Somit können wir $\frac{\partial c}{\partial w_{ij}^{(n-1)}}$ auch so schreiben:

$$\frac{\partial c}{\partial w_{ij}^{(n-1)}} = \delta_j^{(n-1)} \frac{\partial s_j^{(n-1)}}{\partial w_{ij}^{(n-1)}} = \delta_j^{(n-1)} \cdot x_i^{(n-2)}.$$

Allgemein können wir die Ableitung eines Gewichts der *k*-ten Schicht so berechnen:

$$\frac{\partial c}{\partial w_{ij}^{(k)}} = \frac{\partial c}{\partial s_j^{(k)}} \frac{\partial s_j^{(k)}}{\partial w_{ij}^{(k)}} = \delta_j^{(k)} \cdot x_i^{(k-1)}.$$

Dabei hat $\delta_j^{(k)}$ (mit Ausnahme von den $\delta_j^{(n)}$'s der letzten Schicht) den folgenden Wert:

$$\delta_j^{(k)} = \frac{\partial c}{\partial s_j^{(k)}} = \frac{\partial c}{\partial x_j^{(k)}} \frac{\partial x_j^{(k)}}{\partial s_j^{(k)}} = \sum_l (\delta_l^{(k+1)} w_{jl}^{(k+1)}) \cdot a'\left(s_j^{(k)}\right).$$

8.1.5 Änderung der Gewichte

Für die Änderung $\Delta w_{ij}^{(k)}$ gilt nun:

$$\Delta w_{ij}^{(k)} = -\alpha \cdot \frac{\partial c}{\partial w_{ij}^{(k)}} = -\alpha \cdot \delta_j^{(k)} \cdot x_i^{(k-1)}.$$

8.2 Probleme mit linearen Aktivierungsfunktionen

Im Text wird erwähnt, dass lineare Funktionen als Aktivierungsfunktion nicht geeignet sind. Der Grund liegt in der Tatsache, dass man Netze mit linearer Aktivierungsfunktion immer auf ein Netz mit nur einer Schicht Neuronen zurückführen lassen. Um dies zu zeigen, verwende ich dieselbe Notation wie im vorherigen Abschnitt.

Wir zeigen zuerst, dass wir bei Einsatz einer linearen Aktivierungsfunktion die ersten zwei Schichten auf eine Schicht reduzieren können. Daraus folgen wir, dass wir *alle* Schichten auf eine Schicht reduzieren können.

Wir betrachten zu Beginn die Berechnung der Ausgabewerte Schicht für Schicht. Die Ausgabewerte der Neuronen der ersten Schicht lauten wie folgt:

$$x_r^{(1)} = a\left(\sum_{q=1}^{n^{(0)}} x_q^{(0)} \cdot w_{qr}^{(1)}\right).$$

Da wir die Aktivierungsfunktion $a(x) = x$ verwenden, können wir diesen Ausdruck vereinfachen:

$$x_r^{(1)} = a\left(\sum_{q=1}^{n^{(0)}} x_q^{(0)} \cdot w_{qr}^{(1)}\right) = \sum_{q=1}^{n^{(0)}} x_q^{(0)} \cdot w_{qr}^{(1)}.$$

$x_r^{(1)}$ ist der Ausgabewert des r-ten Neurons der ersten Schicht. Wir bewegen uns jetzt eine Schicht nach vorne und berechnen den Ausgabewert eines Neurons der zweiten Schicht:

$$x_s^{(2)} = a\left(\sum_{r=1}^{n^{(1)}} x_r^{(1)} \cdot w_{rs}^{(2)}\right) = \sum_{r=1}^{n^{(1)}} x_r^{(1)} \cdot w_{rs}^{(2)}.$$

Setzen wir für $x_r^{(1)}$ den obigen Ausdruck ein:

$$x_s^{(2)} = \sum_{r=1}^{n^{(1)}} x_r^{(1)} \cdot w_{rs}^{(2)} = \sum_{r=1}^{n^{(1)}} \left(\sum_{q=1}^{n^{(0)}} (x_q^{(0)} \cdot w_{qr}^{(1)}) \cdot w_{rs}^{(2)}\right).$$

$w_{rs}^{(2)}$ können wir in die Summe $\sum_{q=1}^{n^{(0)}} x_q^{(0)} \cdot w_{qr}^{(1)}$ verschieben, da es mit jedem Summanden multipliziert wird:

$$x_s^{(2)} = \sum_{r=1}^{n^{(1)}} \left(\sum_{q=1}^{n^{(0)}} (x_q^{(0)} \cdot w_{qr}^{(1)}) \cdot w_{rs}^{(2)}\right) = \sum_{r=1}^{n^{(1)}} \left(\sum_{q=1}^{n^{(0)}} (x_q^{(0)} \cdot w_{qr}^{(1)} \cdot w_{rs}^{(2)})\right).$$

Da für Summen das *Kommutativgesetz* und das *Assoziativgesetz* gilt, können wir die beiden Summenzeichen vertauschen:

$$x_s^{(2)} = \sum_{r=1}^{n^{(1)}} \left(\sum_{q=1}^{n^{(0)}} (x_q^{(0)} \cdot w_{qr}^{(1)} \cdot w_{rs}^{(2)})\right) = \sum_{q=1}^{n^{(0)}} \left(\sum_{r=1}^{n^{(1)}} (x_q^{(0)} \cdot w_{qr}^{(1)} \cdot w_{rs}^{(2)})\right).$$

Jetzt können wir $x_q^{(0)}$ aus der inneren Summe ausklammern, da er für jeden Summanden der Summe $\sum_{r=1}^{n^{(1)}} (x_q^{(0)} \cdot w_{qr}^{(1)} \cdot w_{rs}^{(2)})$ konstant ist:

$$x_s^{(2)} = \sum_{q=1}^{n^{(0)}} \left(\sum_{r=1}^{n^{(1)}} (x_q^{(0)} \cdot w_{qr}^{(1)} \cdot w_{rs}^{(2)})\right) = \sum_{q=1}^{n^{(0)}} \left(x_q^{(0)} \cdot \sum_{r=1}^{n^{(1)}} (w_{qr}^{(1)} \cdot w_{rs}^{(2)})\right).$$

Nun ist ersichtlich, dass jeder Ausgabewert der Neuronen der zweiten Schicht *linear* von den Eingabewerten $x_q^{(0)}$ des Netzes abhängt. Bezeichnen wir den Linearfaktor mit $\beta_{qs}^{(2)}$:

$$x_s^{(2)} = \sum_{q=1}^{n^{(0)}} \left(x_q^{(0)} \cdot \sum_{r=1}^{n^{(1)}} (w_{qr}^{(1)} \cdot w_{rs}^{(2)})\right) := \sum_{q=1}^{n^{(0)}} x_q^{(0)} \cdot \beta_{qs}^{(2)}.$$

Schreiben wir diesen Ausdruck noch um:

$$x_s^{(2)} = \sum_{q=1}^{n^{(0)}} x_q^{(0)} \cdot \beta_{qs}^{(2)} = a\left(\sum_{q=1}^{n^{(0)}} x_q^{(0)} \cdot \beta_{qs}^{(2)}\right).$$

Jetzt sehen wir, dass wir die Ausgabewerte der Neuronen der zweiten Schicht auch berechnen können, wenn wir uns die erste Schicht wegdenken. Das Gewicht $\beta_{qs}^{(2)}$, welches

dann zur Verbindung vom q-ten Neuron der ersten Schicht zum s-ten Neuron der zweiten Schicht gehört, hat den folgenden Wert:

$$\beta_{qs}^{(2)} := \sum_{r=1}^{n^{(1)}} (w_{qr}^{(1)} \cdot w_{rs}^{(2)}).$$

Ohne, dass das Netz andere Ausgabewerte ausgibt, dürfen wir also die erste Schicht eliminieren. Dies können wir danach gleichermassen mit den restlichen Schichten tun. Schlussendlich können wir jeden Ausgabewert des Netzes als Linearkombination der Eingabewerte berechnen:

$$x_s^{(n)} = \sum_{q=1}^{n^{(0)}} x_q^{(n-1)} \cdot \beta_{qs}^{(n)}.$$

$\beta_{qs}^{(n)}$ können wir ausgehend von $\beta_{qs}^{(2)}$ iterativ berechnen:

$$\beta_{qs}^{(k)} = \sum_{r=1}^{n^{(k-1)}} (w_{qr}^{(k-1)} \cdot w_{rs}^{(k)}).$$

Somit haben wir gesehen, dass sich alle Netze mit einer linearen Aktivierungsfunktion auf ein Netz mit nur einer Schicht Neuronen zurückführen lassen. Doch weshalb ist dies ein Problem?

Netze aus einer Schicht und mit einer linearen Aktivierungsfunktion können bereits einfache Probleme nicht mehr lösen. Ein berühmtes Beispiel ist der XOR-Operator (siehe [39]): Stellen wir uns vor, wir wollen herausfinden, ob von zwei Werten einer der beiden gleich 1 ist. Falls dies der Fall ist, sollte ein Netz 1 ausgeben, in allen anderen Fällen 0:

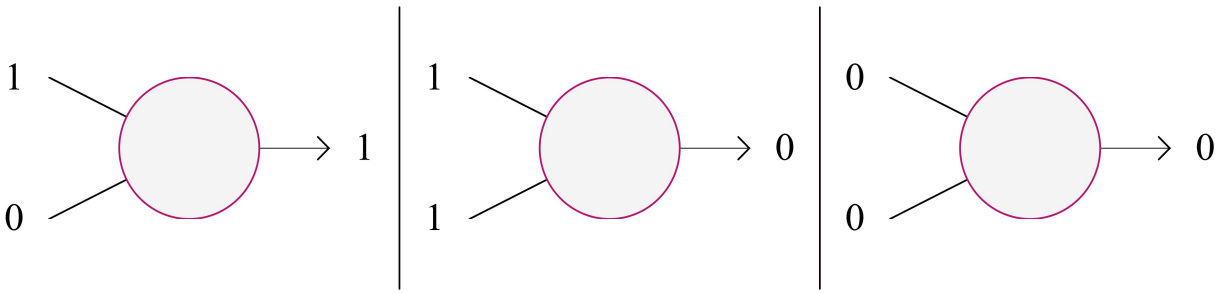


Abbildung 53: Die gewünschten Ausgabewerte bei einem XOR-Operator.

Versuchen wir nun, die passenden Gewichte für ein Netz zu finden, welches dies herausfinden kann und eine lineare Aktivierungsfunktion benutzt.

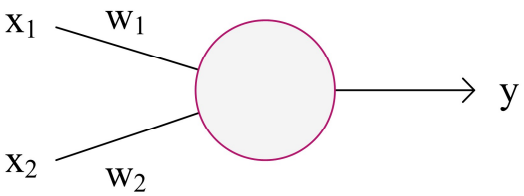


Abbildung 54: Unser Netz, mit welchem wir einen XOR-Operator nachbilden möchten

Der Ausgabewert lässt sich wie folgt berechnen:

$y = x_1 w_1 + x_2 w_2$.

Wir kennen den Ausgabewert für folgende Eingabewerte:

x_1	x_2	y	
0	0	0	$0 \cdot w_1 + 0 \cdot w_2 = 0$
1	0	1	$1 \cdot w_1 + 0 \cdot w_2 = 1$
0	1	1	$0 \cdot w_1 + 1 \cdot w_2 = 1$
1	1	0	$1 \cdot w_1 + 1 \cdot w_2 = 0$

Wir besitzen ein Gleichungssystem aus vier Gleichungen und zwei Variablen. Dieses Gleichungssystem besitzt aber keine Lösungen – die letzten drei Gleichungen sind widersprüchlich. Wir können ein solches Netz mit einer linearen Aktivierungsfunktion nicht realisieren.

8.3 Implementation eines neuronalen Netzes

Um weitere Netztypen implementieren zu können, habe ich mich entschieden, das gesamte Netz in eine eigene Klasse auszulagern. Die Klasse besitzt die folgenden Methoden:

Methodennamen	*Beschreibung*
__init__(size, activationFunction, activationDerivative, costFunction)	Erstellt ein Netz mit einer bestimmten Grösse, einer bestimmten Aktivierungsfunktion und einer bestimmten Fehlerfunktion.
getOutput(inputValues)	Berechnet die Ausgabewerte für bestimmte Eingabewerte.
getDerivatives(outputDerivatives)	Trainiert das Netz anhand der partiellen Ableitungen *outputDerivatives* der Ausgabewerte nach dem Fehler. Die Gewichte und Schwellenwerte werden aber noch nicht angepasst, die Änderungen werden nur gespeichert. Gibt die partiellen Ableitungen der Eingabewerte nach dem Fehler zurück.
trainNetwork(inputValues, correctOutputValues)	Trainiert das Netz. Die Gewichte und Schwellenwerte werden aber noch nicht angepasst, die Änderungen werden nur gespeichert.
applyChanges(learningRate, biasLearningRate, regularization)	Passt die Gewichte und Schwellenwerte mithilfe der gespeicherten Änderungen an.

8.4 Generieren der Sonogramme

Die Sonogramme generiere ich mit dem kostenlosen Tool *sox*. Die mp3-Dateien von Spotify besitzen zwei Kanäle. Man könnte nun aus beiden Kanälen ein separates Sonogramm erstellen. Ich habe mich aber entschieden, zuerst die beiden Kanäle zu einem Kanal zu mischen. Ich kann dies mit dem folgenden Befehl erledigen:

```
sox <inputfile> <outputfile> remix 1,2
```

<inputfile> ist dabei der Dateipfad der Zweikanal-Datei, *<outputfile>* der Dateipfad der erstellten Einkanal-Datei.

Anschliessend erstelle ich die Sonogramme mit folgendem Befehl:

```
sox <inputfile> -n spectrogram -r -m -w Bartlett -y 513 -o <outputfile>
```

<inputfile> ist hier der Dateipfad der Musikdatei des Songs, *<outputfile>* der Dateipfad des erstellten Sonogramms.

8.5 Herleitung der abgeänderten Trio-Fehlerfunktion

Als ich die Trio-Fehlerfunktion (11) aus dem Abschnitt 6.4.1 genauer untersucht habe, ist mir ein Detail aufgefallen, welches eine Schwachstelle darstellen könnte. In diesem Abschnitt möchte ich die Schwachstelle aufzeigen und meinen Lösungsansatz vorstellen.

Die ursprüngliche Trio-Fehlerfunktion (11) sieht wie folgt aus:

$$c(y_A, y_B, y_C) = \frac{1}{2}\sum_i \left(y_{Ai} - y_{Bi}\right)^2 - \frac{1}{2}\sum_i \left(y_{Ai} - y_{Ci}\right)^2.$$

Stellen wir uns im Folgenden ein Netz mit nur einem Ausgabewert vor. Wir berechnen den Ausgabewerte y_A des Netzes bei Eingabe des Songs A und passen anschliessend die Gewichte und Schwellenwerte an.

Wir teilen das Ändern der Gewichte in zwei Schritte auf: Zuerst ändern wir die Gewichte so, dass sich der Ausgabewert y_A von dem Ausgabewert y_C von Song C entfernt. Erst in einem zweiten Schritt ändern wir sie so, dass sich der Ausgabewert von Song A an den Ausgabewert von Song B y_B annähert.

8.5.1 Fehlerfunktion für Songs unterschiedlicher Playlists

Bei der Fehlerfunktion (11) sorgt der zweite Term für das Entfernen des Ausgabewerts y_A von y_C. Nennen wir diesen Term c_{AC}:

$$c_{AC} = -\frac{1}{2}(\mathrm{y_A} - \mathrm{y_C})^2.$$

Betrachten wir ein Gewicht w. Wie müssen wir w bei Verwendung der Fehlerfunktion c_{AC} nun anpassen? Für die Änderung Δw gilt ja bekanntlich:

$$\Delta w = -\alpha \cdot \frac{\partial \mathrm{c_{AC}}}{\partial w}.$$

Mit der Kettenregel erhalten wir:

$$\Delta w = -\alpha \cdot \frac{\partial \mathrm{c_{AC}}}{\partial w} = -\alpha \cdot \frac{\partial c_{AC}}{\partial y_A} \frac{\partial y_A}{\partial w}.$$

Wie stark w angepasst wird, hängt also von der Lernrate, der Ableitung des Fehlers nach dem Ausgabewert und der Ableitung des Ausgabewertes nach dem Gewicht ab. Die Lernrate und $\frac{\partial y_A}{\partial \mathrm{w}}$ sind unabhängig von der gewählten Fehlerfunktion. Anders allerdings $\frac{\partial c_{AC}}{\partial y_A}$, weshalb wir diese genauer unter die Lupe nehmen.

Mit Einsetzen folgt:

$$\frac{\partial \mathrm{c}_{AC}}{\partial \mathrm{y}_A} = \frac{\partial}{\partial \mathrm{y_A}} \left(-\frac{1}{2} (y_A - \mathrm{y_C})^2 \right) = -\frac{1}{2} \cdot 2(y_A - \mathrm{y_C}) \cdot 1 = -(y_A - \mathrm{y_C}) = y_C - y_A.$$

Dieser Term beschreibt den Einfluss des ersten Ausgabewertes auf die Änderung des Gewichts w. Ich nenne ihn $t(y_A)$:

$$t(y_A) := \frac{\partial \mathrm{c}_{AC}}{\partial \mathrm{y}_A} = y_C - y_A.$$

Je grösser der Betrag von $t(y_A)$ ist, desto stärker wird w geändert.

Der Graph von $t(y_A)$ sieht wie folgt aus:

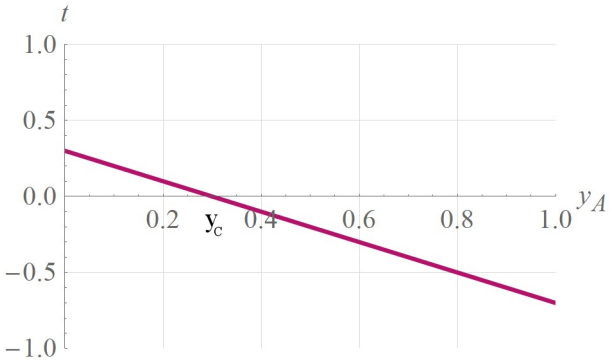


Abbildung 55: t in Abhängigkeit von y_A bei der Trio-Fehlerfunktion des Papers.

Das Gewicht w wird somit am stärksten geändert, falls y_A bereits weit von y_C entfernt ist.

Wäre es aber nun nicht optimaler, wenn das Gewicht am stärksten geändert werden würde, falls y_A in der Nähe von y_C ist? Und falls y_A bereits weit entfernt von y_C ist, das Gewicht nicht mehr stark beeinflusst wird? Ich schlage deshalb eine andere Fehlerfunktion vor, bei der der Graph von $t(y_A)$ wie folgt aussieht:

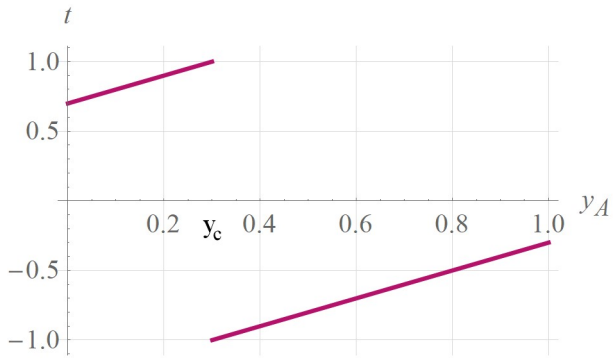


Abbildung 56: t in Abhängigkeit von y_A bei der abgeänderten Trio-Fehlerfunktion.

Die zu diesem Graph führende Funktion lautet $t(y_A) := y_A - y_C - \frac{y_A - y_C}{|y_A - y_C|}$. Wir müssen nun noch die dazugehörige Fehlerfunktion c_{AC} bestimmen.

Wir haben $t(y_A)$ definiert als $t(y_A) := \frac{\partial c_{AC}}{\partial y_A}$:

$$t(y_A) := \frac{\partial c_{AC}}{\partial y_A} = y_A - y_C - \frac{y_A - y_C}{|y_A - y_C|}.$$

An c_{AC} gelangen wir, indem wir diesen Term nach y_A integrieren:

$$c_{AC} = \int \frac{\partial c_{AC}}{\partial y_A} d y_A = \int (y_A - y_C - \frac{y_A - y_C}{|y_A - y_C|})\, d y_A = \frac{y_A^2}{2} - y_A y_C - |y_A - y_C|.$$

Wichtig anzumerken ist, dass diese Fehlerfunktion nur anwendbar ist, solange die Ausgabewerte zwischen 0 und 1 liegen, so wie dies bei Verwendung der Sigmoid-Aktivierungsfunktion der Fall ist. Dies sehen wir im Graphen von c_{AC} in Abhängigkeit zu y_A:

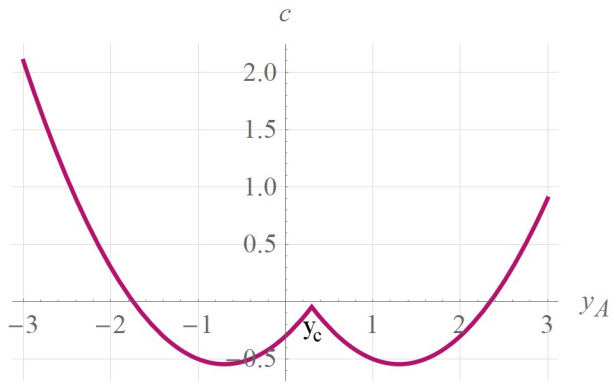


Abbildung 57: c in Abhängigkeit von y_A bei der Trio-Fehlerfunktion des Papers. Ausserhalb von $0 \leq y_A \leq 1$ kann der Fehler wieder steigen.

Ausserhalb von $0 \leq y_A \leq 1$ kann der Fehler trotz zunehmendem Abstand zu y_C wieder steigen.

Für mehrere Ausgabewerte sieht die Fehlerfunktion so aus:

$$y_{AC} = \sum_i \left(\frac{y_{A_i}^2}{2} - y_{A_i} y_{C_i} - \left|y_{A_i} - y_{C_i}\right|\right).$$

8.5.2 Fehlerfunktion für Songs derselben Playlist

Bei der Fehlerfunktion (11) ist der erste Term für das Annähern der Ausgabewerte zweier Songs verantwortlich. Besteht bei diesem Term die gleiche Problematik?

$$c_{AB} = \frac{1}{2}(x_A - y_B)^2.$$

$t(y_A)$ hat in diesem Fall den folgenden Wert:

$$t(y_A) := \frac{\partial c_{AB}}{\partial y_{A_1}} = \frac{\partial}{\partial y_{A_1}}\left(\frac{1}{2}(y_A - y_B)^2\right) = \frac{1}{2} \cdot 2(y_A - y_B) \cdot 1 = y_A - y_B.$$

$t(y_A)$ hat, abgesehen vom Vorzeichen, die gleiche Form wie jenes t bei der Fehlerfunktion bei Songs unterschiedlicher Playlists. Somit können wir die selbe Beobachtung machen: Das Gewicht wird am stärksten geändert, falls y_A weit von y_B entfernt ist.

Dies ist in diesem Fall gewünscht. Wir können diesen Teil der Fehlerfunktion somit beibehalten.

8.5.3 Abgeänderte Trio-Fehlerfunktion

Wenn wir c_{AB} und das neue c_{AC} miteinander kombinieren, erhalten wir:

$$c(y_A, y_B, y_C) = \frac{1}{2}\sum_i \left(y_{Ai} - y_{Bi}\right)^2 + \sum_i \left(\frac{y_{A_i}^2}{2} - y_{A_i} y_{C_i} - \left|y_{A_i} - y_{C_i}\right|\right).$$

8.6 Details zu den verwendeten Parametern

In den folgenden Abschnitten sind die verwendeten Netzparameter der Netze, deren Resultate in der Arbeit erwähnt werden, aufgeführt.

Die Eigenschaft *Netzarchitektur* wird wie folgt angegeben:

Bei den Architekturen ohne Indikatornetze gibt die erste Zahl die Anzahl der Eingabewerte wieder. Die folgenden Zahlen stehen für die Anzahl Neuronen in den einzelnen Schichten des Netzes. *784 – 10* stellt also ein Netz dar, welches 784 Eingabewerte entgegennimmt und eine Schicht mit 10 Neuronen besitzt.

Bei den Architekturen mit Indikatornetzen wird eine Schicht aus Indikatornetzen mit einem *i*, gefolgt von der Anzahl Indikatornetze angegeben. *i50 – 100 – 13* gibt also ein Netz an, welches zuerst aus einer Schicht mit 50 Indikatornetzen besteht. Deren gemittelten Ausgabewerte werden anschliessend einem Netz mit zwei Schichten mit 100 und 13 Neuronen übergeben.

8.6.1 MNIST

8.6.1.1 Anzahl Trainingsbeispiele

Anzahl Trainingsbeispiele	*Anzahl Testbeispiele*	*Netzarchitektur*	*Lernrate α*	*λ*	*Epochen*
100	10'000	784 – 10	1	0.00002	100
1'000					
10'000					
60'000					

8.6.1.2 Lernrate

Anzahl Trainingsbeispiele	*Anzahl Testbeispiele*	*Netzarchitektur*	*Lernrate α*	*λ*	*Epochen*
60'000	10'000	784 – 10	0.001	0.00002	100
			0.01		
			0.1		
			1		
			10		

8.6.1.3 Netzgrösse

Anzahl Trainingsbeispiele	*Anzahl Testbeispiele*	*Netzarchitektur*	*Lernrate α*	*λ*	*Epochen*
60'000	10'000	784 – 10	1	0.00002	100
		784 – 10 – 10			
		784 – 50 – 10			
		784 – 100 – 10			
		784 – 200 – 10			

8.6.1.4 Was lernt das Netz?

Anzahl Trainingsbeispiele	*Anzahl Testbeispiele*	*Netzarchitektur*	*Lernrate α*	*λ*	*Epochen*
60'000	10'000	784 – 10	1	0.00002	100

8.6.2 Genre-Klassifikation

Anzahl Trainingsbeispiele	*Anzahl Testbeispiele*	*Netzarchitektur*	*Lernrate α*	*Epochen*
14'000 (3s-Auschnitte)	100 (ganze Songs)	i50 – 100 – 13	2	300
		i50 – i50 – 100 – 13		
		i50 – i50 – i50 – 100 – 13		

8.6.3 Playlisterstellung

Methode	*Netzarchitektur*	*Lernrate α*	*λ*	*Epochen*
Trio-Fehlerfunktion	250 – 100 – 20	0.00333	0.000005	400
Trio-Fehlerfunktion, abgeändert		0.0333		
Quartett-Fehlerfunktion		1		
Autoencoder				
Autoencoder mit Quartett-Fehlerfunktion				
Cross-Playlist-Encoder				
Cross-Playlist-Encoder mit Quartett-Fehlerfunktion				

9 Quellenverzeichnis

1. Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton: ImageNet Classification with Deep Convolutional Neural Networks, 2012
2. Mateusz Malinowski, Marcus Rohrbach, Mario Fritz: Ask Your Neurons: A Neural-based Approach to Answering Questions about Images, 2015
3. Richard Zhang, Phillip Isola, Alexei A. Efros: Colorful Image Colorization, 2016
4. Jürgen Markl (Ernst Klett Verlag): Markl Biologie, 2010
5. Spektrum der Wissenschaft: Lernen (http://www.spektrum.de/lexikon/biologie/lernen/39008) [abgerufen am 27.08.2017]
6. Spektrum der Wissenschaft: Hebbsche Lernregel (http://www.spektrum.de/lexikon/biologie/hebbsche-regel/30973) [abgerufen am 27.08.2017]
7. Spektrum der Wissenschaft: Langzeitpotenzierung (http://www.spektrum.de/lexikon/biologie/langzeitpotenzierung/38177) [abgerufen am 27.08.2017]
8. Wikipedia: Donald O. Hebb (https://de.wikipedia.org/wiki/Donald_O._Hebb) [abgerufen am 30.10.2017]
9. Ian Goodfellow, Yoshua Bengio, Aaron Courville (The MIT Press): Deep Learning, 2017
10. Neural Networks and Deep Learning: Neural Networks and Deep Learning (http://neuralnetworksanddeeplearning.com/) [abgerufen am 02.06.2017]
11. Wikipedia: Künstliches neuronales Netz (https://de.wikipedia.org/wiki/Künstliches_neuronales_Netz) [abgerufen am 02.06.2017]
12. Xavier Glorot, Antoine Bordes, Yoshua Bengio: Deep Sparse Rectifier Neural Networks, 2011
13. Wikipedia: Sigmoidfunktion (https://de.wikipedia.org/wiki/Sigmoidfunktion) [abgerufen am 02.06.2017]
14. Wikipedia: Euklidischer Abstand (https://de.wikipedia.org/wiki/Euklidischer_Abstand) [abgerufen am 02.12.2017]
15. Wikipedia: Gradientenverfahren (https://de.wikipedia.org/wiki/Gradientenverfahren) [abgerufen am 02.06.2017]
16. Machine Learning Mastery: Overfitting and Underfitting With Machine Learning Algorithms (https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/) [abgerufen am 18.10.2017]
17. Statistics How To: Regularization (http://www.statisticshowto.com/regularization/) [abgerufen am 27.08.2017]
18. ImageNet (http://www.image-net.org/) [abgerufen am 18.10.2017]
19. Minyoung Huh, Pulkit Agrawal, Alexei A. Efros: What makes ImageNet good for transfer learning?, 2016
20. A. Töscher, M. Jahrer: The BigChaos solution to the Netflix Prize 2008. Technical report, commendo research & consulting, 2008

21. Python (https://www.python.org/) [abgerufen am 18.08.2017]
22. LeCun: MNIST (http://yann.lecun.com/exdb/mnist/) [abgerufen am 08.07.2017]
23. Wikipedia: Joseph Fourier (https://de.wikipedia.org/wiki/Joseph_Fourier) [abgerufen am 26.11.2017]
24. Spotify Web API (https://developer.spotify.com/web-api/) [abgerufen am 19.10.2017]
25. SoX - Sound eXchange (http://sox.sourceforge.net/) [abgerufen am 19.10.2017]
26. Sander Dieleman: Recommending music on Spotify with deep learning (http://benanne.github.io/2014/08/05/spotify-cnns.html) [abgerufen am 09.10.2017]
27. Chatbots Life: Finding the genre of a song with Deep Learning (https://chatbotslife.com/finding-the-genre-of-a-song-with-deep-learning-da8f59a61194) [abgerufen am 09.10.2017]
28. Keras: The Python Deep Learning library (https://keras.io/) [abgerufen am 29.10.2017]
29. Spotify: What is a Good Playlist? (https://community.spotify.com/t5/Music-Chat/What-is-a-Good-Playlist/td-p/1173531) [abgerufen am 19.10.2017]
30. Nicole Cifani: Playlist Manifesting: What Makes a Great Mixtape? (http://nicolecifani.com/2010/08/playlist-manifesting-what-makes-a-great-mixtape/) [abgerufen am 19.10.2017]
31. Unicum: So geht's: Die perfekte Playlist für jede Stimmung erstellen (https://www.unicum.de/de/entertainment/musik/so-gehts-die-perfekte-playlist-fuer-jede-stimmung-erstellen) [abgerufen am 19.10.2017]
32. Robert O. Gjerdingen, David Perrott: Scanning the Dial: The Rapid Recognition of Music Genres, 2008
33. Wikipedia: Musikgenres (https://de.wikipedia.org/wiki/Kategorie:Musikgenre) [abgerufen am 27.07.2017]
34. M. Sordo, O. Celma, M. Blech, E. Guaus: The Quest for Musical Genres: Do the Experts and the Wisdom of Crowds Agree?, 2008
35. Techcrunch: Inside The Spotify – Echo Nest Skunkworks (https://techcrunch.com/2014/10/19/the-sonic-mad-scientists/) [abgerufen am 27.07.2017]
36. Raia Hadsell, Sumit Chopra, Yann LeCun: Dimensionality Reduction by Learning an Invariant Mapping, 2005
37. Florian Schroff, Dmitry Kalenichenko, James Philbin: FaceNet: A Unified Embedding for Face Recognition and Clustering, 2015
38. Wikipedia: Autoencoder (https://de.wikipedia.org/wiki/Autoencoder) [abgerufen am 12.10.2017]
39. Wikipedia: XOR-Gatter (https://de.wikipedia.org/wiki/XOR-Gatter) [abgerufen am 08.07.2017]
40. Neural Networks and Deep Learning: How the backpropagation algorithm works (http://neuralnetworksanddeeplearning.com/chap2.html) [abgerufen am 11.11.2017]

10 Redlichkeitserklärung

Ich bestätige hiermit, dass ich meine Maturaarbeit selbständig und ohne unerlaubte Mithilfe verfasst und keine anderen als die angegebenen Quellen benutzt habe.

Oberhallau, 24.03.2018

Tobia Ochsner